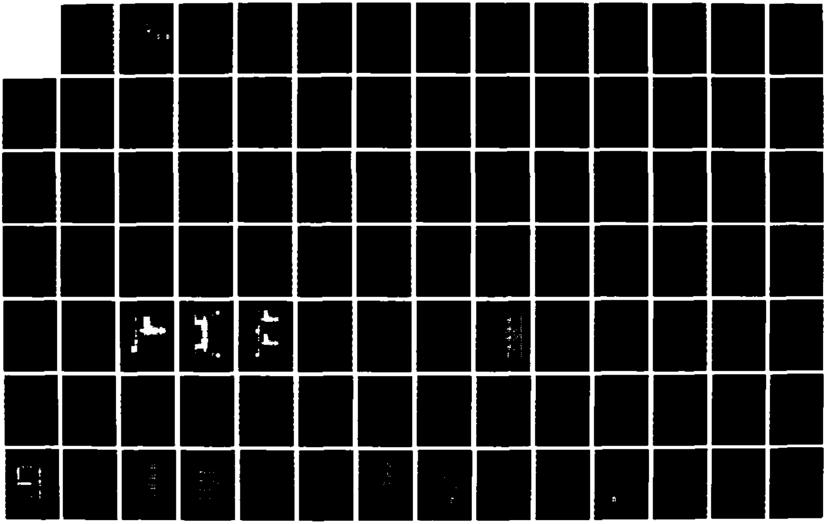
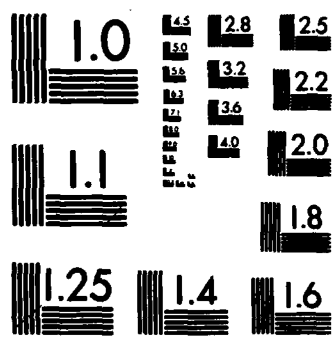
AD-A171 369 SILICON COMPILATION USING A LISP-BASED LAYOUT LANGUAGE 1/2
(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA
M A MALAGON-FAJAR JUN 86

UNCLASSIFIED F/G 9/2 NL






MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California



# THESIS

SILICON COMPILATION USING A LISP-BASED LAYOUT LANGUAGE

by

Manuel Ambrosio Malagon-Fajar

June 1986

Thesis Advisor: D. E. Kirk

Approved for public release; distribution is unlimited.

SECURITY CLASSIFICATION OF THIS PAGE

AD-A171 369

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited. |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4 PERFORMING ORGANIZATION REPORT NUMBER(S) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School; 6b. OFFICE SYMBOL (If applicable) 62 | 7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School |
| 6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000 | 7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000 |
| 8a NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c ADDRESS (City, State, and ZIP Code) | 10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO |

11 TITLE (Include Security Classification)
SILICON COMPILATION USING A LISP-BASED LAYOUT LANGUAGE

12 PERSONAL AUTHOR(S)
Manuel Ambrosio Malagon-Fajar

| 13a TYPE OF REPORT | 13b TIME COVERED | 14 DATE OF REPORT (Year, Month, Day) | 15 PAGE COUNT |
|---|---|---|---|
| Master's Thesis | FROM ____ TO ____ | 86 June | 208 |

16 SUPPLEMENTARY NOTATION

| 17 COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Silicon Compilation; LISP; Layout Language |
| | | | |
| | | | |

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Two related silicon compilers developed at MIT's Lincoln Laboratory with a common layout language are examined. The simpler one, the Lincoln Boolean Synthesizer (LBS), is a Complementary Metal Oxide (CMOS) technology based program for generating chips out of arbitrary boolean expressions. MacPitts, on the other hand, can implement advanced programming language constructs in N-Channel (NMOS) technology. A study of their layout language, Lincoln Laboratory's LISP-based Layout Language (L5), and its implementation is presented. In addition, there is also a brief discussion of how Macpitts's functional repertoire can be changed.

| | | |
|---|---|---|
| 20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS | 21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED | |
| 22a NAME OF RESPONSIBLE INDIVIDUAL Prof D. E. Kirk | 22b TELEPHONE (Include Area Code) (408)646-3451 | 22c OFFICE SYMBOL 62Ki |

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

## ABSTRACT

Two related silicon compilers developed at MIT's Lincoln Laboratory with a common layout language are examined. The simpler one, the Lincoln Boolean Synthesizer (LBS), is a Complementary Metal Oxide (CMOS) technology based program for generating chips out of arbitrary boolean expressions. MacPitts, on the other hand, implements advanced programming language constructs in N-Channel (NMOS) technology. A study of their layout language, Lincoln Laboratory's LISP-based Layout Language (L5), and its use is presented. In addition, there is also a brief discussion of how Macpitts' functional repertoire can be changed.

## LIST OF FIGURES

# ACKNOWLEDGEMENT

Many thanks to Professors D. Kirk, H. Loomis and B. MacLennan for valuable time spent discussing this thesis and related ideas.

Professors R. Hamming and R. Strum, with their iconoclastic manner, were a cheerful source of encouragement.

MIT Lincoln Laboratories provided a copy of LBS and MacPitts to work with. Their research in this area was a source of motivation.

Dr. A. Domic and Dr. D. Johannsen through their lectures, writings and personal contact answered many questions and fostered an interest in programs that make computers.

Special appreciation is extended to B. Limes, D. Shaffer, A. Wong, S. Whalen, M. Williams and R. Johnson for their technical expertise and support.

Dr. F. A. Malagón-Díaz and Mrs. E. Fajar de Malagón provided many years of loving nurture and the belief that ideas shape the world.

CAPT E. Malagón, USMC, read through several drafts and gave many constructive comments.

LT A. Mullarky, USN, created the equality cell which was used to modify a MacPitts' functional unit.

MAJ E. Weist, USMC, worked on a graphical interface to MacPitts and gave useful insights into the compiler's more arcane aspects.

This thesis is dedicated to the American people and Navy.

Modern philosophers like Dreyfus, Haugeland, Heidegger, Husserl and Wittgenstein take different stances on what constitutes intelligence.[2]

In the meantime, success in war and peace depends on computers. Sensors, controllers and actuators melded into smart machines build cars round the clock or kill at long range. Additionally, computing machines process data used in all phases of decision making. The range of use extends from simple word-processors up to expert consultants.

However, the potential use of computers has only begun to be explored. And, though there have been many impressive results from computer expert systems, they have been limited to specific domains of expertise. Therefore, in order to break through to a new level of processing activity, the Defense Advanced Research Projects Agency (DARPA) launched a major Strategic Computing (SC) program. (DARPA, 1983, pp 1-18)

SC has a goal of creating a widespread machine intelligence technology in the United States. It aims at creating a prototype autonomous land vehicle, a pilot's associate and a battle management system. The SC program is multi-level and addresses issues from microelectronics to software design. However, several areas, such as vision and speech recognition, which humans do so effortlessly, are difficult for machines with present approaches as indicated in this quote (DARPA, 1983, p. 33):

> Recent progress in developing vision for navigation has been severely constrained by lack of adequate computing hardware. Not only are the machines which are now being used too large to be carried by the experimental vehicles, but current machines are far too slow to execute the vision algorithms in real-time . . . .

---

[2] See the bibliography.

The first problem is addressed by creating a flowchart interface in which the user graphically creates state diagrams that are converted for the user into MacPitts programs. (Weist, 1986)

The second issue is the subject of this thesis: an examination of Lincoln Laboratory's LISP based Layout Language (L5) and its relation to MacPitts. L5 is a LISP based language used by MacPitts to compile Very Large Scale Integrated (VLSI) circuits automatically. L5 is also used by the Lincoln Boolean Synthesizer (LBS), a Complementary Metal Oxide Semiconductor (CMOS) compiler of arbitrary boolean expressions, to generate combinational logic circuits.

Both of these compilers have many interacting programs linked together to execute automatically. Alteration of this behaviour requires that the programs, composed of L5 and LISP code, be modified.

Therefore, the main questions examined in this thesis are:

- How is L5 created?
- How is L5 used?

The answer to these questions is given by:

- Introducing LISP;
- Covering LISP extensions needed to create L5 (lincoln.l);
- Presenting L5;
- Grouping several programs into a "compiler"; and,
- Modifying a MacPitts functional unit.

LISP fundamentals are covered in Chapter II. The ideas of functional programming and other general concepts are discussed. After this overview, the presentation covers LISP functions and usage. Additionally, a look is

Appendix A contains a description of alignment problems caused by incorrect CIF plotting or organelle specification; and, a sketch of how to experiment in the MacPitts environment.

In summary, this thesis covers L5, a flexible idiom for procedurally creating VLSI circuits, and shows how understanding L5 makes MacPitts and LBS accessible for modification.

1. Functional Programming

*Imperative languages* are based on directing control through a series of assignment statements. LISP on the other hand applies functions to their arguments. (MacLennan, 1983, p. 345)

A function takes a combination of arguments and assigns a unique value to it. A *functional* or *applicative* language[1] is built upon a simple idea that is well illustrated in this quote (Hofstadter, 1985, p. 452) :

> A programmer's instinct says that you can cumulatively build a system, encapsulating all the complexity of one layer into a few functions, then building the next layer up by exploiting the efficient and compact functions defined in the preceding layer. This hierarchical mode of buildup would seem to allow you to make arbitrarily complex actions be represented at the top level by very simple function calls.

This spirit of functional application pervades both MacPitts and LBS. But, before looking at LISP's functions, a language for talking about LISP, Backus Naur Format, is introduced.

2. Backus Naur Format (BNF)

BNF is a concise set of symbols for describing the syntax of computer languages. Its key idea is that the description should look like the language it's talking about (MacLennan, 1983, pp. 166-173). A terse set of BNF symbols is given below:

- The " < " and " > " indicate syntactic categories. For example, <integer>, <LISP form>, etc..
- The " ::= " means " is defined as ".

---

[1] Haugeland, 1984, pp. 125-164 gives a very cogent explanation of several computer architectures [LISP included].

There is another important LISP object, a **list**, defined below:

<list>[3] ::= **(**<atom>***)** | **(**{<atom> | <list>}***)**

A **list** is a left parenthesis followed with zero or more atoms or lists, closed off with a right parenthesis. Notice that this is a recursive definition: a <list> is defined in terms of itself. Examples of lists are:

**()**, **(a)**, **(a b (c d) e)**.

Note that **()**, **nil**, is both an **atom** and a **list**.

BNF is used throughout this thesis to describe LISP syntax. LISP's basic functional format, <lambda function>, can now be analyzed.

3. Lambda Functions

One method for writing functions in LISP is with lambda notation. [For other function definition formats see Section II.C.2] Perhaps the easiest way to understand lambda notation is with this quote showing its history (Touretzky, 1984, p. 86):

> Lambda notation was created by Alonzo Church, a mathematician at Princeton University, as an unambiguous way of specifying functions, their inputs, and the computations they perform. In lambda notation, a function that added 3 to a number would be written $\lambda x.(3 + x)$. The $\lambda$ is the Greek letter lambda.

---

[3] Refer to Sections II.C.1 and II.C.3.b. A list can also be viewed in this light:

<list> ::= **(**<head><tail>**)**
<head> ::= { <atom> | <list> }
<tail> ::= <list>

*For example:*

| | | |
|---|---|---|
| **()** | has <head> := **nil** | and <tail> := **nil** |
| **(a)** | has <head> := **a** | and <tail> := **nil** |
| **(a b (c d) e)** | has <head> := **a** | and <tail> := **(b (c d) e)** |

The lambda function format can be named by using the LISP primitive **def**[6] in this manner:

```
<function-name> ::=
                  -> (def <function-name> <lambda function>) <CR>
<function-name> ::= <atom>
```

A function created with **def** is applied to its argument's parameters by using its name as follows:

```
<value> ::= -> (<function-name><parameter>*)<CR>
```

By naming the function, its usefulness is increased. Instead of typing the unwieldy lambda form each time the function is applied, the user simply types in the function's name. Consider $F(x,y) = 3x + y2$ defined as a LISP function named **quadratic**:

```
-> (def quadratic
;; <function-name> ::= -> (def <function-name><lambda function>)
     (lambda (x y)
        (plus (times 3 x)(times y y))))<CR>
;; LISP returns <function-name>:
   quadratic
```

This function, **quadratic**, is applied by using its name with parameters:

```
-> (quadratic 2 3)<CR>
;; (quadratic <x><y>)
      15

-> (quadratic (quadratic -1 2)(quadratic 2 3))<CR>
;; (quadratic -1 2) := 1 & (quadratic 2 3) := 15
;; (quadratic 1 15) := 228
      228
```

---

[6] See Section II.C.2 for another method for defining functions [ **defun** ].

**static variable:** a variable that is allocated before execution of the program begins and that remains allocated for the duration of execution of the program.

**variable ([illegible])([illegible])**

(1) in computer programming, a character or group of characters that refers to a value and, in the execution of a computer program, corresponds to an address.

(2) a quantity which can assume any of a given set of values . . . .

Three more terms need to be defined: A *bound variable* is one of a function's formal parameters [function's arguments]. A *global variable* has its value set at the top level. A *free variable* is not a bound variable, but its value is used or changed by a function. (Wilensky, 1984, pp. 39-40) Now that the terms have been defined, the concept of variable scoping can be examined.

There are two basic variable scoping techniques "-- static scoping and dynamic scoping. In static scoping (also called lexical scoping) a procedure is called in the environment of its definition; in dynamic scoping a procedure is called in the environment of its caller." (MacLennan, pp. 112-113, 1983). In other words, (MacLennan, p. 109, 1983):

- In dynamic scoping the meanings of statements and expressions are determined by the dynamic structure of the computations evolving in time.

- In static scoping the meanings of statements and expressions are determined by the static structure of the program.

Franz LISP is a dynamically scoped language.[7] Therefore, bound variables which are changed during a function call are restored to their original values upon exiting the function. If calls to other functions are

---

[7] COMMON LISP is a lexically scoped language (Winston, 1984, p. 54).

```
-> (defun[9] test (bound)
       (setq bound (1+ bound))
;; bound := bound + 1
;; The symbol "free" is not bound within the context of test.
       (+ bound free))<CR>
;; the result := bound + free
         test

-> (test free)<CR>
;; First, "bound" assumes "free's" value: bound := free's value := 2
;; Second, bound := bound + 1 := 2 + 1 := 3
;; Third, the result := bound + free := 3 +2  = 5
         5
```

In contrast if LISP used *call by reference*:

(1) bound ::= free := 2

(2) "bound" increments: bound := bound + 1 := 1 + 2 = 3

(3) since bound ::= free, "free" also becomes 3: free := 3

(4) the result is: bound + free := 3 + 3 = 6

In summary, Franz Lisp resolves the problems of variable context and scoping by using call by value and dynamic scoping. This issue can be extended to functions. Next, consider how functions refer to other functions or to themselves.

5. Recursion and Iteration

LISP allows functions to refer to themselves. This approach, known as recursion, is briefly introduced in this section.[10] Suppose a function that raises a given integer base to a nonnegative integer power is desired. Two

---

[9] **defun** is an alternate method of defining functions, see Section II.C.2.

[10] A more in depth discussion of recursion is given in Section II.C.4.

## B. INTERPRETED, COMPILED OR DUMPED LISP

The interpreter allows interactive running of LISP programs and provides an effective environment for debugging LISP code. At the same time, LISP also provides a compiler which can considerably speed up program execution for large code segments. This section examines the different ways LISP can be run and covers very basic input and output.

### 1. The LISP Read-Eval-Print Loop: Interpreted LISP

In Section II.A.1, several examples showed how the LISP interpreter "reads" and "evaluates" input, and then "prints" out a result. This read-eval-print loop is discussed in this section. The two major participants in this cycle, **eval** and **quote**, are also covered.

#### a. The LISP Prompt " -> ", Start " ( " and Stop " ) "

To obtain " -> ", so that <LISP form>s with " ( " and " ) " can run, the Franz Lisp interpreter is invoked by typing **lisp** after the UNIX® prompt:

```
% lisp<CR>
Franz Lisp Opus 38.69
->
```

The " -> " is a prompt sign which means that inputs will be "evaluated" or "interpreted". An open parenthesis, " ( ", instructs the interpreter to do whatever follows, and a closed parenthesis, " ) ", tells the interpreter to stop doing it. (Wilensky, 1984, p. 2)(Hasemer, 1984, p. 6) Therefore, if the user inputs: **(plus 1 2 3)** <CR>, the " ( " starts the LISP interpreter "plusing" **1** with **2**, then with **3**, and stops "plusing" upon reaching " ) ". For example:

```
-> (plus 1 2 3) <CR>
   6
```

```
                      <LISP form>s[13] or
                        <expression>s
                          /        \
                     <atom>s        <list>s
                     /     \
             <number>s   <symbol>s
              /      \
<floating-point>   <integer>s
```

Figure 2.2 The LISP Object Hierarchy

Examples of these LISP objects are shown in Table 2.1:

TABLE 2.1

EXAMPLES OF LISP OBJECTS

| LISP Objects | Example LISP Code |
| --- | --- |
| • <LISP form>s | **(plus 1 2), 1.23, (* (plus 1 2) 3),** . . . |
| • <list>s | **(), (((q w)(e) 1 2)), (plus 1 2),** . . . |
| • <atom>s | **1, 1.1, a, woman,** . . . |
| • <symbol>s | **a, man, we223, %%err,** . . . |
| • <special symbol>s | " ' ", " \ ", " , ", " ; ", " ,@ "[14] . . . |

It seems that LISP is always searching for a value. The next section answers the question: "How does it accept something literally?"

c. Eval's dual: **quote** or " ' "

When evaluation is undesirable it is inhibited with **quote** or its abbreviated form, a quote mark. The " ' " is a <special symbol> that stops evaluation. This idea is evident from the syntax:

<LISP form> ::= -> {(**quote** <LISP form>) | '<LISP form>}<CR>

---

[13] Refer to Sections II.A.2, II.A.3 and II.C.1.

[14] "\" [backslash] is an escape character. " , " and " ,@ " are described in Section III.A.1.2

However, there are errors where execution might not be stopped by the interpreter[15]. In that case, LISP can be stopped with an interrupt. The first control C [ **^C** ] sets an interrupt flag: the system waits for a "safe" place to exit. The second **^C** forces all system calls to compiled code to check the interrupt flag; and finally, a third **^C** causes an immediate interrupt. (Foderado, 1983, Section 10.6) Here is an example:

```
^C ^C ^C
Interrupt: ^C
Break nil
<1>:
```

An interpreter is a useful interactive tool; however, to handle large programs and obtain efficient object code, a compiler is needed.

2. Compiled LISP

Compilation of LISP programs increases their execution speed. In order to keep compilation dependencies among several programs straightened out, a makefile is used. In addition, a makefile can join together several programs so they can run as a large unit.

a. The Compiler

The Franz Lisp compiler is invoked from the UNIX® C-Shell with the following command (Foderado, 1983, Chapter 12):

**% liszt** [-<option>*] <filename>

There are several options, among which, **q** [compile in quiet mode] and **x** [create a cross reference file] are very useful. The compiler can be run with several options at one time as follows:

---

[15] Richard Hamming has jokingly said that perhaps computers do in fact show free will, it's just that people always call a repairman when they do it.

Consider the following example makefile composed of four dependent results [**L5.o**, **work**, **clean**, and **doc**]. The desired results are separated by a colon from their prerequisites and placed on the same line. Notice that two results, **clean**, and **doc**, have no associated prerequisites. The next line contains the actions to create each result. Assume that all of this code is in a file named " Makefile " in the user's directory which contains L5.l and lincoln.l. Makefile's contents are now presented, and described immediately afterwards [the explanation continues into Section II.B.3:

```
L5.o: L5.l lincoln.o
    liszt -qa L5

work: L5.o lincoln.o
    echo[17] "(eval-when (eval)\[18]
        (load 'lincoln.o)(load 'L5.o)\
        (dumplisp[19] work)(exit))" | lisp

clean: rm[20] -f L5.o lincoln.o
```

---

[17] The **echo** command prints out its arguments. The function, **eval-when**, tells the LISP compiler to evaluate the expressions that follow, instead of compiling them. (Wilensky, 1983, 281)

[18] The backslash " \ " is an escape character, therefore the next line is treated as a continuation. The " | " stands for "pipe", i.e., the results of the first process are passed on to the next process.

[19] Saves the LISP environment in an executable file named "work". Typing "work" will then recreate the LISP system as it was running when it was dumped.

[20] Forced removal of files.

```
Examples      array.l            top.l
INSTALL       bin                extract.l
L5.l          c-routines.c       sim.l
Makefile
```

* And finally, back to the UNIX® prompt.

```
%
```

LISP files which are dependent on each other can be organized using a makefile. They can also be individually loaded into the interpreter and saved as one executable file using **dumplisp**.

3. Interpreted and Compiled LISP: **dumplisp**

In some programming languages disparate programs can be combined to form a working unit using a linker. In LISP this can be achieved by creating an "environment" that contains all the programs. This is what the **work** section of the example makefile created in the previous section does:

```
% make work<CR>
```

* Execute the actions under the "work" heading of this makefile.

```
    echo "(eval-when (eval)(load 'lincoln.o)(load 'L5.0)
        (dumplisp work)(exit))" | lisp
```

* Load lincoln.o, L5.o and organelles.o into LISP, dump this envi-
*ronment in an executable file named "work" and then exit LISP.

```
    Franz Lisp, Opus 38.69
-> [fasl lincoln.o]
```

;; fasl is the function LISP uses to load object code files.

```
-> [fasl L5.o]
%
```

In summmary, an executable file, **work**, has been created. Typing **work** as an imperative command places the user in LISP with the functions in L5 and lincoln also available.

```
% work<CR>
->
```

```
% cat .lisprc<CR>
; The UNIX® "cat" command dumps the file ".lisprc" onto the
; terminal screen.
    (eval-when (load eval)
        (load 'lincoln.o)
        (load 'L5.o)
        (load 'organelles.o) )
```

Since LISP automatically loads the .lisprc file [in this case all that the file contains is one large **eval-when** <LISP form>], then the result is that all three **load** functions are evaluated and the files loaded in.

```
% lisp<CR>
; The lisp interpreter is invoked and the .lisprc file is loaded.
Franz Lisp Opus 38.69
->
```

The user is now in LISP with the three files loaded. The main difference between using this method and **dumplisp** is that a dumped file usually requires at least a megabyte of storage, whereas loading several files using the .lisprc file takes a short while.[23] In Chapter V.A and Appendix A.B it will be seen that the MacPitts and LBS environments can be invoked by typing their respective names without any arguments. For example:

```
% macpitts [or lbs]
usage: macpitts <filename> [<options>]
->
```

A closer look is now taken at how files are input into LISP and how functions can be output into files.

---

[23] A compromise between these two approaches is to use the autorun option when compiling a LISP file [e.g., **% liszt -r** <filename>]. This creates an object file which has a small piece of bootstrap code attached. The object file can then be run as an executable file. (Wilensky, 1984, p. 284)

The pretty print function can also be used to send <LISP form>s to a file in the following fashion:

-> **(pp**[26] **(F temp.l) m-to-the-n)**<CR>
;; Output the function m-to-the-n to the file temp.l.
**t**

Conversely, a <LISP form> can be read from a file, without being evaluated, using **read**:

-> **(read (infile 'temp.l))**<CR>
;; Read the next <LISP form> from the temp.l file. When the end of
;; file is reached then nil is returned. The <LISP form> is not
;; evaluated when read. To do so eval must be explicitly used. For
;; example: (eval (read (infile '4-flags))), where 4-flags has a
;; <LISP form> that needs to be evaluated.
**(def m-to-the-n**
**(lambda (m n)**
**(cond ((zerop n) 1)**
**(t (times m (m-to-the-n m (1- n)))) ) ) )**

-> **(exit)**<CR>
;; Leave LISP and then output temp.l to the screen using cat

---

[26] Other functions that are used for output are **patom** and **print**. Their syntax is similar:

<LISP form> ::=
-> **(patom** ['']<LISP form> [**(outfile** <filename> **['a])]**)<CR>
<LISP form>::=
-> **(print** ['']<LISP form> [**(outfile** <filename> **['a])]**)<CR>

These functions both output to the terminal if the optional outfile argument is not given [the **'a** appends the output to the previous file contents, otherwise they are wiped out]. Because these functions do not send carriage returns when they finish their output, they are usually seen in conjunction with **(terpri [(outfile** <filename> **['a])** which outputs a terminate line character sequence. For example:

-> **(patom '| Stop printing. |)(terpri)**<CR>
**Stop printing.**

1. LISP's Basic Structure: The List

A function and a list of data look the same in LISP. For example, the next <LISP form>,

**(replace-item-points inverter new-points),**

is an application of a function [**replace-item-points**] to its arguments [**inverter** and **new-points**]; or, it can also be a list of three elements [**replace-item-points**, **inverter** and **new-points**].

Which one it it? It is both! A LISP program is a list, and **eval** normally applies the list's head as a function to the list's tail. If the list is quoted, then it's treated as data. (MacLennan, 1983, p. 348)

Atoms and lists are referred to as symbolic *expressions*. Expressions are called *forms* if they are to be evaluated. "Considered as data, a list may be called an expression; considered as a piece of procedure, the same list may be called a form". (Winston, 1984, p. 20)

With these ideas in mind another look can be taken at the procedure for LISP function definition.

2. LISP Function Definition: **def** and **defun**[27]

Up to this point the reader has seen functions that take a fixed number of arguments all of which are evaluated. This class of functions is called an *expr*. There are three other categories: *fexpr*, *lexpr* and *macros*[28]. An fexpr takes an unlimited number of arguments, but does'nt evaluate them.

---

[27] See Section II.A.2 for function definition using **def**.

[28] Macros are discussed in Chapter III.

For example, a function that finds the logarithm base 2 of a number can be defined in LISP as follows:

```
-> (defun log-two (number &optional (base 2))
;; The primitive LISP function quotient finds the quotient of two
;; numbers, and log finds the natural logarithm of a number. The
;; optional argument "base" defaults to a value of 2 if a
;; parameter is not given for it.
     (quotient (log number)(log base)) )<CR>
;; Find the logarithm base 2 or the given base of a number.
          log-two
```

This function is applied in the following ways:

```
-> (log-two 13)<CR>
;; (log-two <number>)
;; Find the log base two [defaul] of 13.
     3.700439718141092

-> (log-two 13 10)<CR>
;; Evaluate the base ten log of 13.
     1.113943352306837
```

Another way to define this lexpr is as follows:

```
-> (defun log-two n
;; In this format, the symbol "n", will be bound with the number
;; of arguments supplied. The function arg gives the parameter
;; associated with the position corresponding to the number it is
;; given.
     (quotient
          (log (arg 1))
;; If a second parameter is provided use its value, if not use 2.
          (log (cond
                    ((> n 1)(arg 2))
                    (t 2) ) ) ) )<CR>
        log-two
```

The third functional class, an fexpr, doesn't evaluate its arguments and takes a variable number of them. Nothing comes for free though, the flexibility of a variable number of inputs is offset by the overhead of

## 3. Frequently Used LISP Functions

A synopsis of common LISP functions is presented to briefly familiarize the reader with LISP's syntax. First, a look at functions used to give values to symbols.

### a. Binding Variables: **set**, **setq**, **let** and **let***

Variables are assigned values with **set** or **setq** [**set** **q**uote]. Although **set** only takes one symbol at a time, it has a similar syntax to **setq**:

**(set** {[']<symbol>} {[']<LISP form>}**)**
**(setq** {<symbol> [']<LISP form>}+**)**

These two functions are applied as follows:

-> **(set 'A '(a b c))**<CR>
;; Set "A" to have the value "(a b c)".
    **(a b c)**

-> **A**<CR>
;; A's value is (a b c).
    **(a b c)**

-> **(setq B A  C '(1 2 3) D (plus 1 2 3))**<CR>
;; The <symbol>s are unevaluated, but are respectively assigned
;; the results of evaluating the <LISP form>s. setq returns the
;; value of the last evaluation it performs.
;; B := A, C := (1 2 3) and D := (plus 1 2 3) := 6
    **6**

-> **B**<CR>
;; B's value has been set to A, but A := (a b c).
    **(a b c)**

-> **C**<CR>
;; C's value is (1 2 3).
    **(1 2 3)**

The variables are restored to the values they had prior to participating in the **let*** construct. With these methods of variable assignment in hand, a look is now taken at list manipulation.

b. List Selection: **car** ,**cdr**[33], **nth** , and **nthcdr**

LISP is based on the application of functions to arguments. The syntax of LISP generally has a structure of the form:

**(**<function-name><argument>***)**

Therefore, it seems natural to have a selector that picks the first element of a list, the "function", and another selector that returns all the elements of a list except the first, the "arguments". These selectors are **car** and **cdr**:

```
<head> ::= -> (car <list>)<CR>
<tail> ::= -> (cdr <list>)<CR>
<list> ::= (<head><tail>)[34]
<head> ::= <LISP form>
<tail> ::= <LISP form>
```

The application of these basic selector functions is shown below:

```
-> (car '(plus 1 2 3 4))<CR>
;; (car <list>)
;; car selects the first ("function" or "head") list element
     plus
```

The " tail" selector, **cdr**, is used as follows:

---

[33] **car** and **cdr** were assembly language instructions for the IBM 704 on which LISP was first implemented. An instruction was divided up into fields. Two of the fields were named the *address* and *decrement*. **car** and **cdr** were the instructions for getting the contents of the address pointed to by these fields. (Charniak, 1985, p.48)

[34] Compare to the definition of a list in Section II.A.2.

**-> (nth 3 '(and in these days it came to pass))**<CR>
;; (nth <index><list>)
;; Starting at 0, return the indexed argument of the given list.
**days**

**->(nthcdr 2 '(hylomorphism: all is form & matter))**<CR>
;; (nthcdr <index><list>)
;; Starting at 0, return the indexed cdr of the given list.
**(form & matter)**

Lists can be separated into their components with the functions covered in this section; but, how are they built up?

c. List Construction: **cons**, **append** and **list**

The list selectors **car** and **cdr** separate a list into its "head" or "function" and its "tail" or "arguments". The list constructor **cons** is their dual: it synthesizes a "head" and "tail" into a list. (Winston, 1984, p. 29-31)

<list> ::= **-> (cons** [']<head> [']<tail>**)**<CR>
<list> ::= **(**<head><tail>**)** ::= **-> (cons** '<head> '<tail>**)**<CR>
<head> ::= <LISP form>, and <tail> ::= <list>[35]

Therefore, in order to synthesize a list out of two parts:

**-> (cons 'plus '(1 2 3))**<CR>
;; (cons '<head> '<tail>)
**(plus 1 2 3)**

To create lists use **list** with this format:

<list> ::= **-> (list** { [']<LISP form> }***)**<CR>

An example that makes a list out of several arguments is:

**-> (list 'This 'is 'a 'joined 'sentence!)**<CR>
;; Make a list out of the following elements.
**(This is a joined sentence!)**

---

[35] In actuality an atom can form the tail element, this produces a dotted list, e.g., **(**<head>**.**<tail>**)**

```
-> (apply 'append '((a b)(c d)(e f)))<CR>
   (a b c d e f)
```

**funcall** is similar to apply, except that it accepts each parameter for the function individually. It has this format:

<value> ::= **-> (funcall** <function-name> { [']<parameter> }***)**<CR>

Examples of **funcall** now follow:

```
-> (funcall 'plus 1 2 3)<CR>
;; (funcall <function> { [']<parameter>}* )
      6

-> (funcall 'append '(a b) '(c d) '(e f))<CR>
   (a b c d e f)
```

Up to this point, functions can be applied sequentially to each other; but so far, there is no way to conditionally apply a function. In order to build control structures that can do this, the idea of a predicate is now introduced.

e. Predicates (the Values **t** and **nil**) and the **cond** Control Structure

A predicate is a function whose value is either true or false. The LISP symbol for true is **t** and for false it's **nil**. In LISP any non-**nil** value is considered to be true. Both **t** and **nil** evaluate to themselves. The empty list is also called **nil** and is the only LISP expression that is simultaneously a list and an atom! (Winston, 1984, p. 44-46)

Therefore, the following is true:

{ **t** | **nil** } ::= **-> (**<predicate><LISP form>***)**<CR>

Many LISP predicates end with a **p**, e.g. **listp**, **minusp**, etc., but there are important exceptions such as: **atom**, **null** and **equal**. (Touretzky, 1984, pp. 14-17) So, for example:

```
-> (and)<CR>
;; If "and" has no arguments it returns t.
    t

-> (and 1 2 (plus 2 3))<CR>
;; If all its arguments are non-nil, then "and" gives the value of
;; its last argument; otherwise, if any argument evaluates to nil
;; the result is nil.
    5

-> (or)<CR>
;; If "or" has no arguments it returns nil.
    nil

-> (or (zerop[38] 1)(* 3 5)())<CR>
;; Returns the first non-nil value, otherwise if all its
;; arguments evaluate to nil, " or " returns nil.
    15
```

In another example, examine how a predicate, **member?**,[39] is constructed using conditional tests and the LISP function **member**:

```
-> (member 'a '(b c a d e))<CR>
  ;; member returns a list that starts with the first instance
  ;; of the element that is being checked for membership in a
  ;; list.
  (a d e)
```

The code for the **member?** predicate is now shown. Observe that " list " is a parameter and not the **list** function:

---

[38]
```
-> (zerop 1)<CR>
nil
-> (zerop 0)<CR>
t
```

[39] See Chapter III for a description of lincoln.l. In lincoln.l predicates usually end with a " **?** ".

```
-> (defun match-that
     (thing list predicate &optional tail)
      (cond
        ;; This is the recursion's basis condition:
        ;; If the list is empty, then all the results are in the tail.
        ;; Since the first elements are being consed into the tail
        ;; first by the application of match-that to the remainder
        ;; of the list, (cdr list), when the basis condition is met,
        ;; all the element in the tail will be backwards.
        ;; Therefore, reverse them and return this as the result.
        ;; This is the Basis Condition: stop if the list is empty.
        ((null? list)(reverse tail))
        ;; The list wasn't empty, therefore, apply the predicate
        ;; to the element's head. If the predicate is satisfied,
        ;; place the head in the list called "tail".
        ;; This is a Recursive Condition: apply the predicate to
        ;; first list element, (car list), and match-that to the
        ;; rest of the list, (cdr list).
        ((funcall predicate thing (car list))
         (match-that
              thing
              (cdr list)
              predicate
              (cons (car list) tail)) )
        ;; Since the list wasn't empty and the head element did'nt
        ;; satisfy the predicate, apply this algorithm to the rest
        ;; of the list. Another Recursive Condition.
       (t
        (match-that thing (cdr list) predicate tail][40]<CR>
;; LISP returns the function's name
        match-that
```

Predicates can also be used in iterative control structures.

---

[40] The " ] " is a right superparenthesis. A right superparenthesis can substitute for as many regular parenthesis, " ) " as would be required to close off the <LISP form>. However, the count stops as soon as a left superparenthesis, " [ ", is encountered. (Wilensky, 1985, p. 42)

The **setq**'s are used to assign values to variables within the context of the **prog**. As an example, review this definition of a factorial function:

```
-> (defun factorial (integer)
;; Bind local variables to nil.
     (prog (result)
;; Initialize local variables
       (setq result 1)
;; A loop that will find the factorial of a positive integer.
      loop
;; IF the integer is zero then exit the prog and return the result.
         (cond ((zerop integer)(return result)))
;; OTHERWISE, multiply the integer by the accumulated result,
;; then decrement the integer by one and repeat the loop.
         (setq result (* integer result))
         (setq integer (1-  integer))(go loop) ) )<CR>
    factorial
```

A more structured iterative syntax, which can do everything **prog** does, uses **do** or **do*** (Winston, 1984, p. 86):

```
(do ({(<variable> <initial-value> <update-form>)}*)
     ( <end-test> <LISP form>* <result-form>) <body> )[42]
<end-test> ::= <test form>[43]
<result-form> ::= <LISP form>, and <body> ::= <LISP form>*
```

However, if an action is to be performed across lists, then "the lazy man's do loop", **mapcar**, can be used. (Winston, 1984, p. 79) For example, given the LISP primitive **zerop**, a list's elements can all be checked for equality with zero in one fell swoop:

```
-> (mapcar 'zerop '(1 0 a 0 0 2))<CR>
               (nil t nil t t nil)
```

---

[42] See Section II.C.4 for an example of **do**.

[43] See Section II.C.3.e for <test form>'s format.

```
-> (defun m-to-the-n (m n)
     (do[45] ((result 1 (* m result))
          (power n (- power 1)))
         ((zerop power) result)))<CR>
;; Raise a number to a positive power: m^n.
   m-to-the-n

-> (m-to-the-n 2 3)<CR>
   ;; result1 := 1, power := 3, (zerop 3) := nil
   ;; result2 := (* 2 1) := 2, power := (- 3 1) := 2, (zerop 2) := nil
   ;; result3 := (* 2 2) := 4, power := (- 2 1) := 1, (zerop 1) := nil
   ;; result4 := (* 2 4) := 8, power := (- 1 1) := 0, (zerop 0) := t
      8
```

Recursion accomplishes indefinite repetition "by having a function call itself during its execution." (Wilensky, 1984, p. 73) A recursive implementation of **m-to-the-n** (Winston, 1984, p. 64):

```
-> (defun m-to-the-n (m n)
   ;; The exponent [ n ] should be a non-negative integer.
   (cond
     ;; Test to see if the exponent [ n ] is zero,
     ;; if it is, return a value of one.
     ;; This is the Basis Condition.
     ((zerop n) 1)
     ;; If the exponent is not one, then
     ;; multiply m by (m-to-the-n m (1- n)), n.b.,
     ;; the recursion will end since n will be reduced
     ;; to zero and (m-to-the-n m 0) is one!
     ;; This is the Recursive Condition.
     (t (* m (m-to-the-n m (1- n)[46]))) ) ) )<CR>
     m-to-the-n
```

---

[45] Refer to Section II.C.3.e for **do**'s syntax.

[46] **1-** decrements by one, while **1+** increments by one.

logical error. The three basic functions associated with these programs are **trace**, **debug** and **step**.[47] To see how they work, recall **factorial**:

```
-> (factorial[48] 5)<CR>
;; 5*4*3*2*1 := 120
     120
```

The operation of **zerop** can be observed using **trace**, as follows:

```
-> (trace zerop)<CR>
[autoload /usr/lib/lisp/trace]
[fasl /usr/lib/lisp/trace.o]
;; The tracer returns a list of functions being traced.
     (zerop)
```

Now, every time that **zerop** is used its associated values are shown:

```
-> (factorial 5)<CR>
1 <Enter> zerop (5)
1 <EXIT>  zerop  nil
1 <Enter> zerop (4)
1 <EXIT>  zerop  nil
1 <Enter> zerop (3)
1 <EXIT>  zerop  nil
1 <Enter> zerop (2)
1 <EXIT>  zerop  nil
1 <Enter> zerop (1)
1 <EXIT>  zerop  nil
1 <Enter> zerop (0)
1 <EXIT>  zerop  t
120
```

---

[47] For discussions of these areas see:
(Foderado, 1983, Chapter 11 [Tracer], Chapter 14 [Stepper], Chapter15 [Debugger] and Chapter 16 [Editor])
(Wilensky, 1984, Chapter 11 [Debugging])
(Charniak, 1985, Section 2.8 [Debugging])
(Winston, 1984, Chapter 14 [Debugging])

[48] Defined in Section II.C.3.f.

```
(setq integer (|1-| integer))n2<CR>
2
(go loop)
(cond ((zerop integer) (return result)))
  (zerop integer)d<CR>
;; Go into debug mode. Usually invoked with: (debug)
[fasl /usr/lib/lisp/fix.o]

<------debug------>
;; Obtain a listing of debug commands using help.
: help<CR>
```

| | |
|---|---|
| u / u n / u f / u n f | go up, i.e. more recent (n frames) (of function f) |
| up / up n | go up to next (nth) non-system function |
| d / dn | go down, i.e. less recent (opposite of u and up) |
| ok / go | continue after an error or debug loop |
| redo / redo f | resume computation from current frame (or at fn f) |
| step | restart in single-step mode |
| return e | return from call with value of e (default is nil) |
| edit | edit the current stack frame |
| editf / editf f | edit nearest fn on stack (or edit fn f) |
| top / bot | go to top (bottom) of stack |
| p / pp | show current stack frame (pretty print) |
| where | give current stack position |
| help / h / ? | print this table -- /usr/lisp/doc/fixit.ref |
| help ... | get the help for ... |
| pop / ^d | exit one level of debug (reset) |

```
funcall-evalhook*
evalhook*
(zerop integer)
(cond (<**> (return result)))
evalhook
continue-evaluation
funcall-evalhook*
evalhook*
(cond ((zerop integer) (return result)))
(prog (result) (setq result 1) loop ...)
evalhook
continue-evaluation
funcall-evalhook*
evalhook*
(prog (result) (setq result 1) loop ...)
(factorial 3)
evalhook
continue-evaluation
funcall-evalhook*
evalhook*
(factorial 3)
;; The stack has LISP system function calls interspersed
;; with the factorial function. A handy feature of the
;; error loop is that the current variable values can be
;; easily obtained. Showstack returns nil.
nil
;; What is the "integer" variable's value?
<1>: integer<CR>
2
;; What is the "result" variable's value?
<1>: result<CR>
3
;; Leave the error loop.
<1>: (reset)<CR>

[Return to top level]
```

Hopefully, this very brief look at some LISP programming tools will encourage the user to experiment with them. The next section reviews the salient points covered up to now.

• Lisp functions (equivalent to subroutines or procedures in other languages) are data objects that can be passed as parameters to other functions. This makes it possible to write extensible control structures in user programs that are very difficult to duplicate in more traditional languages.

1. <u>Data Abstraction and Macros</u>

Abstraction of low level functions can aid understanding. For example, the unmnemonic **car** might be renamed **head**:

-> **(defun head (x)(car x))**<CR>
**head**

-> **(head '(A B C D))**<CR>
**A**

The mnemonic quality of this new function is offset by the overhead of having a user defined function calling a LISP system function. The LISP function **car** takes one instruction, but a user defined function takes five or more instructions! (Brooks, 1984, pp. 179-180)

Since data abstraction is an important programming tool, the cost of the extra function calls in compiled code is removed by the use of macros. "A macro is a function which accepts a Lisp expression as input and returns another Lisp expression." (Foderado, 1983, p. 8-3)

A macro is efficient because it creates code that the LISP interpreter evaluates only once. Subsequent calls to the macro use the expanded code (Wilensky, 1984, pp. 180-195). The function **defmacro** [**def**ine **macro**] is one of three ways to create a macro (Foderado, 1983, p. 8-3). For example:

<macro-name> ::=
-> **(defmacro** <macro-name> **(**<argument>*****)<LISP form>*****)**<CR>

<macro-name> ::=
-> **(def** <macro-name> **(macro (**<argument>**)**<LISP form>*****))**<CR>

<macro-name> ::=
->**(defun** <macro-name> **macro (**<argument>**)**<LISP form>*****)**<CR>

A macro is applied just like a function:

<value> ::= -> **(**<macro-name><parameter>*****)**<CR>

the backquote macro: expressions are not evaluated unless specified. (Foderado, 1983, pp. 8-3,8-4) The symbol for inhibiting evaluation is " $ ", for evaluating " , " and for evaluating and splicing into a list " ,@ ". (Wilensky, 1984, p. 202) These symbols can be applied in succession, as composite operators, and are summarized in Table 3.1 below:

TABLE 3.1

BACKQUOTE MACRO SYMBOLS

| Symbol | Function |
|---|---|
| **$** | Inhibit one level of evaluation |
| **,** | Evaluate [within the context of " **$** "] |
| **,@** | Evaluate and append |
| **$,** or **,$** | No-ops, they can be removed.[4] |
| **,@$( )** or **,@'( )** | No-ops |
| **$(,x)** | **(list x)** |
| **$(,x ,@y)** | **(cons x y)** [y must be a list] |
| **$(,@x ,@y)** | **(append x y)** [x & y must evaluate to lists] |
| **$(,@'x ,@'y)** | **(append 'x 'y)** [x and y must be lists] |

So for example, if the variable **A** is set to have as its value the list **(1 2 3)**, the effect of " **$** ", "**,**" and "**,@**" can be observed:

```
-> (setq A '(1 2 3))<CR>
;; The variable "A" is assigned the list " (1 2 3) " as a value.
        (1 2 3 )

-> $(A ,A ,@A)<CR>
;; " A " is unevaluated, " ,A " is evaluated, " ,@A " is evaluated and
;; spliced into the list structure.
        (A (1 2 3) 1 2 3)
```

---

[4]
```
-> $((a b) (C D) ,@'(e f) ,@'(G H))<CR>
;; " ,@' " acts as a composite operator: ,@(quote (<argument>))
;; So, first apply quote, and then " ,@ ".
((a b) (C D) e f G H)
```

The backquote macro is frequently used in writing macros. It's used to create a template of the code the macro will provide to **eval**, for example:

```
-> (defmacro head (X) $(car ,X))<CR>
;; Equivalent to: (defmacro head (X)(list 'car X))
        head
```

These ideas are all brought to fruition when functions that generate other functions are made. A good example is the **defstruct** [**def**ine **struct**ure] macro.[5] This macro consists of two levels. The lowest level creates the desired function according to a template. The upper level evaluates the function that was created. A brief sketch and a bit of the LISP code demonstrates the idea:[6]

```
                 eval
                  |
              defstruct
              /       \
          eval         eval
          /               \
defstruct-short   defstruct-long
```

Figure 3.1 The Defstruct Function Hierarchy

The code that follows reflects the structure in Figure 3.1. There is a main **eval-when** form that evaluates the **defstruct** function. This function in turn has two **eval-when** forms in it. They will either evaluate

---

[5] See Section III.C for more detail on **defstruct**.

[6] The reader should skim through this code looking at how the evaluation statements are nested with macro or function definitions. Look at the code's form and the extent that it "shows" the macros it is generating. The LISP function **eval-when** tells the interpreter or compiler to evaluate this code when it is loaded into LISP.

As an example, a list with fields "name" and "age" will be called a "man". Examining the results from the bottom up shows how functions are first created and then evaluated into the LISP environment. First the lowest level functions **defstruct-short-fields** and **defstruct-replace-fields** create macro definitions in the following fashion:

```
-> (defstruct-short-fields 'man '(name age) 1)<CR>
   ;; Since there are two fields two selector macro definitions
   ;; are made. They are returned in a list.The results are:
   ;; A macro definition that selects the name field: man-name.
   ((def man-name (macro (body)$(car ,(cadr body))))
   ;; A macro definition that selects the age field: man-age.
   (def man-age (macro (body)$(cadr ,(cadr body)))))

-> (defstruct-replace-fields 'man '(name age) 1)<CR>
   ;; Since there are two fields two mutator macro definitions
   ;; are made. They are returned in a list.
   ;; A macro definition that replaces the name field with a new
   ;;  value is created and called replace-man-name.
   ((def replace-man-name (macro (body)$(append
       (list ,(caddr body))(cdr ,(cadr body)))))
   ;; A macro definition that replaces the age field with a new
   ;;  value is created and named replace-man-age.
   (def replace-man-age (macro (body)$(append
       (list (car ,(cadr body)),(caddr body))
       (cddr ,(cadr body))))) )
```

The above results are now spliced into a list of macros:

```
-> (defstruct-short 'man '(name age))<CR>
   ;; The macro definitions are spliced into a list:
   ((def make-man (macro (body) ... )
    (def man-name (macro (body) ... )
    (def man-age (macro (body)  ... )
    (def replace-man-name (macro (body) ... )
    (def replace-man-age (macro (body) ... ) )
```

```
(def defstruct-long
 (lambda (type body)
  (cond ((null body) ())
        ((or (null (cdr body))
             (list? (car body))
             (atom? (cadr body)))
         (err '|invalid defstruct syntax|))
        (t (append (cons (list 'def
                               (concat 'make- (car body) '- type)
                               (list 'macro
                                     '(body)
                                     (list 'cons
                                           ''list
                                           (list 'cons
                                                 (list 'list
                                                       ''quote
                                                       (list 'quote
                                                             (car body)))
                                                 '(cdr body)))))
                         (cons (list 'def
                                     (concat 'is- type '- (car body) '?)
                                     (list 'macro
                                           '(body)
                                           (list 'list
                                                 ''eq
                                                 (list 'list
                                                       ''car
                                                       '(cadr body))
                                                 (list 'list
                                                       ''quote
                                                       (list 'quote
                                                             (car body))))))
                               (append
                                (defstruct-long-fields
                                 type (car body) (cadr body) 2)
                                (defstruct-replace-fields
                                 (concat (car body) '- type)
                                 (cadr body)
```

Figure 3.3 The **defstruct-long** Definition Without Backquote

called an organelle which is covered in Chapter VI. (Siskind, 1982, pp. 14-15)(Lincoln Lab Report 662, 1983, pp. 25-26)

TABLE 3.2

NUMERICAL COMPARISON PREDICATES

| Predicate Name | Predicate Test |
|---|---|
| **=0** | equality with zero |
| **<0** | negative sign |
| **>0** | positive sign |
| **>=0** | non-negative value |
| **<=0** | non-positive value |
| **<=** | less than or equal |
| **>=** | greater than or equal |
| **<>0** | not equal to zero |
| **<>** | not equal |
| **=1** | equality with one |

In addition to the numerical comparison macros shown above, lincoln.l has several macros that perform type checking.

b. Type Predicate Macros

LISP's applicative nature allows functions to be passed as data and provides data handling flexibility at the expense of performing very little type checking.[8] (Gray, P., 1984, p. 111) Whereas in LISP predicates usually have the form <name>**p**, in linconl.l they have the form <name>**?**. Take for example a LISP and a lincon.l predicate that checks if a number is odd:

```
-> (oddp 3)<CR>
;; LISP predicates often end in a " p ".
t
```

[8] For a discussion of type checking see (Aho, 1986, pp. 343-380).

B. FUNCTIONS

1. APL Like Operators

APL was one of the first programming languages to apply functions over whole data structures, thereby freeing the programmer from the tedium of iterating over elements.[9] John Backus, one of FORTRAN's creators, wanted to reason algebraically about programs and suggested applying APL's ideas in a purely functional manner. The operations of this algebra would consist of applying, binding, selecting, "composing, reversing, mapping and reducing functions." (MacLennan, 1983, p. 405)

Several functions are shown here as examples of the many useful functions with an APL flavor in this section:

```
-> (such-that '( 0 -1 9 -2 -3) '<0)<CR>
;; (such-that <list> <predicate>)
;; Return all list elements satisfying the predicate.
        (-1 -2 -3)

-> (slash '((a b c)(d c a b)(e f g h)) nil 'union)<CR>
;; (slash <list> <identity> <function>)
;; Return the result of applying a function to a list's elements.
        (a b c d e f g h)

-> (sort '(1 4 2 5 3 9) '>)<CR>
;; (sort <list> <predicate>)
;; Sort a list's elements by a predicate.
        (9 5 4 3 2 1)

-> (car-list '((1 2)(3 4)(5 6)))<CR>
;; (car-list <list>)
;; Find the first element of each of a list's sublists.
        (1 3 5)
```

[9] For an excellent APL user's guide see (IBM, 1983, p. 13).

**-> (nthset-list**
**'(1 4 6)**
**'(Sad is the woman who cries along the way.)**
**'(*Happy man sings*)** )<CR>
;; (nthset-list <index-list> <template-list> <new-element-list>)
;; Replace the indexed positions in the template-list with the
;; respective elements from the new-element-list.
**(*Happy* is the *man* who *sings* along the way.)**

3. <u>Set Functions</u>

A LISP list can be viewed as a set with elements, e.g.:

{ element1, element2, ..., elementN } :== **(**<element>***)**
<set> ::= <list>

With this point of view in mind lincoln.l provides a variety of set operators:

**-> (setify '((a b)(a b c)*(a b)*(a) 2 'a *(a)*))**<CR>
;; (setify <list>)
;; Remove redundant elements from a list. Notice that (a b) and
;; (a) occur more than once in the list. Italics are for emphasis.
**((a b c)(a b) 2 'a (a))**

**-> (union '(1 2 3 4) '(2 3 5 4 6 7))**<CR>
;; (union <set>$_1$<set>$_2$)
**(1 2 3 4 5 6 7)**

**-> (intersection '(1 2 3 4 5) '(3 4 5 6))**<CR>
;; (intersection <set>$_1$<set>$_2$)
**(3 4 5)**

**-> (set- '(1 2 3 4 5) '(2 4))**<CR>
;; (set- <set>$_1$ <set>$_2$)
;; Remove set$_2$ elements from set$_1$.
**(1 3 5)**

The wide spectrum of functions seen in this section crop up throughout MacPitts and LBS. Surprisingly enough though, a large portion of the functions encountered in these programs are generated by one macro: **defstruct**.

## C. DEFSTRUCTS[12]

The lincoln.l **defstruct** [**def**ine **struct**ure macro] allows the user to create new data types. It automatically generates macros to create, select, change or type check instances of the data type. The following quote states the idea of a structure (Winston, 1984, p. 100):

> Conceptually, a *structure* is a collection of *fields* and *field values*. We are allowed to define new structures by specifying their particular field names and default field values. We are further allowed to construct individual structures of any already defined type, to access those individual structures, and to revise them. However, in keeping with the spirit of data abstraction, we are not allowed to look at the way individual structuresare represented internally, for we are supposed to be isolated from the actual representation.

**Defstructs** are frequently used throughout LBS and MacPitts. They are a useful tool when a large number of different data types must be manipulated. The **defstruct** macro creates short or long data structures.

### 1. Short Defstructs

The short **defstruct** has the following format:

```
<short form> ::= {<field>+ | { <field>*<list> }}
<field> ::= <symbol>
```

[12] Refer to the examples in Section III.A.2. Lincoln.l's **defstruct** macro is slightly different from those found in other versions of LISP.

Notice that the result is a list with all the field values placed in the order they were entered.

b. Short Selector

**Defstruct** also creates selector macros to obtain field values. A short selector macro that picks out <field>$_i$ of a <type> short defstruct has the format:

<short-selector> ::= <type>-<field>$_i$
<field-value>$_i$ ::=
**-> (**<type>-<field>$_i$ {**'(**<field-value>$^+$**)** |
**(list** [']<field-value>$^+$**)**} **)**<CR>

For example:
**-> (point-name '(in 3 7 NM ((signal)(river))))**<CR>
;; Get the point's name:
**in**

**-> (point-attributes '(uss -2 7 NB ((power)(out))))**<CR>
;; Get the point's attributes:
**((power)(out))**

c. Short Mutator

The third macro automatically generated for a short **defstruct** is used to change field values. Mutators replace a <type> **defstruct**'s <field-value>$_i$ with <field-*new*-value>$_i$ and have the following form:

<short-mutator> ::= **replace**-<type>-<field>$_i$
**(**<field-value>$_1$...<field-*new*-value>$_i$ ...<field-value>$_N$**)** ::=
**-> (replace**-<type>-<field>$_i$
{ **'(**<field-value>$^+$**)** | **(list** [']<field-value>$^+$**)** }
{ [']<field-*new*-value>$_i$ }**)**<CR>

In both the short and long structure cases **defstruct** is used. A long **defstruct** example with a **tree** genus and eight species:

```
-> (defstruct tree
       null ()
       rect (layer left bottom right top)
       symbol-call (name)
       move (tree dx dy)
       rotcw (tree)
       rotccw (tree)
       mirrorx (tree)
       mirrory (tree))<CR>
```

This long structure creates a **tree** data type. There are eight **tree** cases: **null**, **rect**, **symbol-call**, **move**, **rotcw**, **rotccw**, **mirrorx** and **mirrory**[15]. Note that five of the **tree** cases have a **tree** in their field. The field arguments are also **defstructs**! A long structure has four associated functions: constructors, selectors, mutators and interrogators.

a. Long Constructor

As in the short structure, macros to construct data type instances are automatically generated in the long structure. A constructor that instantiates the species <case$_i$>-<type> has this format:

<long-constructor> ::= **make**-<case>$_i$-<type>
**(**<case>$_i$ <<case>$_i$-field-value>***)** ::=
**-> (make**-<case>$_i$-<type> { [']<<case>$_i$-field-value>}+**)**<CR>

---

[15]These eight cases correspond to eight basic operations on rectangles. **Null** is no action or no rectangle. **Rect** is a rectangle with the given layer and dimensions. **Symbol-call** represents a method for generating hierarchical representation. **Move**, **rotcw**, **rotccw**, **mirrorx** and **mirrory** represent respectively a displacement by dx and dy; ninety degree clockwise and counterclockwise rotation; and a flip about the x axis or the y axis. The operators these trees represent are described in (Crouch, 1984, p. 8).

<long-mutator> ::= **replace**-<case>$_i$-<type>-<field>$_j$  
**(**<case>$_i$  
<<case>$_i$-<field>$_1$-value> ...  
<<case>$_i$-<field>$_j$-*new*-value> ...  
<<case>$_i$-<field>$_N$-value> **)** ::=  
**-> (replace**-<case>$_i$-<type>-<field>$_j$  
{'**(**<case>$_i$ <<case>$_i$-field-value>$^+$**)** |  
**(list** { [']<case>$_i$ }{ [']<<case>$_i$-field-value> }$^+$ **)** }  
{[']<<case>$_i$-<field>$_j$-*new*-value>} **)**<CR>

This is best seen in a few examples:

**-> (replace-rect-tree-top '(rect NM 1 2 3 4) 15)**<CR>  
;; replace a "rect-tree" species' "top" field with 15.  
**(rect NM 1 2 3 15)**

**-> (replace-move-tree-dy**  
**'(move (rect 1 2 3 4) 9 8)**  
**11)**<CR>  
;; replace a "move-tree" species' "dy" field with 11.  
**(move (rect 1 2 3 4) 9 11)**

The **tree** example has shown that a long structure adds a level of complexity to the **defstruct** concept. Why bother? Because there is a big advantage to be gained in grouping similar ideas together and then differentiating between them. In order to do this a long **defstruct** also creates interrogators.

d. Long Interrogator

Long structures offer a limited form of data type checking with their interrogator macros. A check to see if a structure is a <case>$_i$-<type> species can be made as follows:

The other check that is made ensures that only the last field in a **defstruct** is a list. This occurs in **defstruct-short-fields** and **defstruct-long-fields** where the fields are also checked to be not empty.

4. Summary

**defstruct** offers the programmer a tool for data abstraction. This idea along with the mnemonic character of constructors, selectors, mutators and interrogators are great aids in data manipulation. **defstruct**s are extensively used in LBS and MacPitts. It might also be speculated that to some degree the mind-body paradigm is reflected in MacPitts' function-data language and controller--data-path architecture. In any case, Table 3.4 presents a **defstruct** summary:

TABLE 3.4

DEFSTRUCT FUNCTION SUMMARY

| Function | Short Structure | Long Structure |
|---|---|---|
| Constructor | **make**-<type> | **make**-<case>$_i$-<type> |
| Selector | <type>-<field>$_i$ | <case>$_i$-<type>-<field>$_j$ |
| Mutator | **replace**-<type>-<field>$_i$ | **replace**-<case>$_i$-<type>-<field>$_j$ |
| Interrogator | None | **is**-<type>-<case>$_i$**?** |

TABLE 4.1

GLOBAL VARIABLES AND THEIR FUNCTIONS

<table>
<tr><th>Variable</th><th>Status Check</th><th>Status Modifier & Options</th></tr>
<tr><td>--L5-symbol-storage</td><td>(L5-symbol-storage)</td><td>(L5-symbol-storage! ['](<on-disk | in-memory>))</td></tr>
<tr><td>--L5-technology</td><td>(technology)</td><td>(technology! ['](<nmos | cmos | cmos-pw | cmos3 | sos| scmos>))</td></tr>
<tr><td>--L5-minimum-feature-size</td><td>(minimum-feature-size)</td><td>(minimum-feature-size! <centi-μ per λ>)</td></tr>
<tr><td>--L5-symbol-list</td><td>(L5-symbol-list) & (create-called-symbol-item <position>)</td><td>(add-symbol-to-L5-symbol-list <symbol>)</td></tr>
<tr><td>--L5-symbol-number</td><td>--L5-symbol-number</td><td>(setq --L5-symbol-number <integer>) & (symbol-number)</td></tr>
<tr><td>--L5-symbol-port</td><td>(L5-symbol-port)</td><td>(setq --L5-symbol-port <port>)</td></tr>
<tr><td>--L5-symbol-file</td><td>(L5-symbol-file)</td><td>(setq --L5-symbol-file <file>)</td></tr>
<tr><td>**</td><td>(allowed-layers)</td><td>**</td></tr>
<tr><td>**</td><td>(allowed-conducting-layers)</td><td>**</td></tr>
<tr><td>**</td><td>(layer-table)</td><td>**</td></tr>
<tr><td>%%</td><td>(allowed-technologies)</td><td>**</td></tr>
</table>

All of the global variables can be changed using **setq**. Functions with, **, operate by checking the technology global variable and returning an appropriate response without setting any variables. The, **, denotes that to change the values returned by these functions the LISP source code has to be

**-> (L5-symbol-file)**<CR>
;; This file is used to output CIF[3] results.
    **/tmp/L5sym12904**

**-> (L5-symbol-port)**<CR>
;; A port is a LISP I/O device.
    **%/tmp/L5sym12904**

**-> (technology! 'cmos)**<CR>
;; Set the technology to cmos and list out its layers.
;;These symbols correspond to CIF layers, e.g. CD = n-type
;;diffusion, CP = polysilicon, CM = first layer metal, etc..
    **(CD CP CM CM2 CS CC CG CW NX XP)**

**-> (technology)**<CR>
;; The current technology is complementary metal oxide
;; semiconductor
    **cmos**

**-> (technology! 'scmos)**<CR>
;;These are Calma scalable cmos CIF layers, e.g. CMS =
;;metal2, CMF = metal1, CPG = polysilicon, etc..
**(CMS CMF CPG CAA CVA CCP CCA CWP CWN
CSP CSN COG)**

**-> (minimum-feature-size! 50)**<CR>
;; Set 50 centimicrons to be 1 lambda unit.
    **50**

**-> (minimum-feature-size)**<CR>
;; Currently 50 centimicrons are 1 lambda unit.
    **50**

---

[3] "The Caltech Intermediate Form (CIF Version 2.0) is a means of describing graphic items (mask features) of interest to LSI circuit and system designers." (Mead, 1980, p. 115) Also see (Sequin, 1980, Chapter 7) and (Scott, 1986, Magic Tutorial #9 and Magic Technology Manual #1-2).

## 2. L5 Data Structures

All L5 data structures are created using **defstruct**. The generic object in L5 is called an **item** and is composed of rectangles and labels. (Crouch, 1983, p. 2) Since an **item** is a grouping of smaller objects it is surrounded with an imaginary rectangle [box] which encompasses all its elements. The smallest box which encloses an **item** is called the Minimum Bounding Box [MBB]. (Ayres, 1983, p. 64) The syntax for an **item** is:

TABLE 4.3

AN ITEM'S SYNTAX

<table>
<tr><th>Category</th><th>Syntax</th></tr>
<tr><td><item> ::=</td><td>(<left><bottom><right><top><points><br><called-symbol-names><tree>)</td></tr>
<tr><td>{<left>|<bottom>|<br><right>|<top>} ::=</td><td><number></td></tr>
<tr><td><points> ::=</td><td>(<point>*)</td></tr>
<tr><td><point> ::=</td><td>(<name><x><y><attributes>)</td></tr>
<tr><td><attributes> ::=</td><td>{ (<symbol>*) | ({(<symbol>)}*) }</td></tr>
<tr><td><called-symbol-<br>names> ::=</td><td>(<number>*)</td></tr>
<tr><td><tree> ::=</td><td>{ <null-tree> | <rect-tree> | <symbol-call-tree> |<br><move-tree> | <rotcw-tree> | <rotccw-tree> |<br><mirrorx-tree> | <mirrory-tree> }</td></tr>
<tr><td><null-tree> ::=</td><td>(null)</td></tr>
<tr><td><rect-tree> ::=</td><td>(rect <layer><left><bottom><right><top>)</td></tr>
<tr><td><symbol-call-<br>tree> ::=</td><td>(symbol-call <number>*)</td></tr>
<tr><td><move-tree> ::=</td><td>(move <tree>)</td></tr>
<tr><td><rotcw-tree> ::=</td><td>(rotcw <tree>)</td></tr>
<tr><td><rotccw-tree> ::=</td><td>(rotccw <tree>)</td></tr>
<tr><td><mirrorx-tree> ::=</td><td>(mirrorx <tree>)</td></tr>
<tr><td><mirrory-tree> ::=</td><td>(mirrory <tree>)</td></tr>
</table>

An **item** structure contains two other structures within it: a list of **point** short structures and a **tree** long structure. First, a look at the **item** structure and the creation of a simple **item**:

As is seen in the above example, a list of points is a field in an item. The **point** short structure is implemented as follows:

```
-> (defstruct point (name x y layer attributes))
;; A point is a label. Points have names, are located at
;; a specific x and y location and are attached to a layer.
;; A point's attributes can give descriptive information
;; to guide functional application. For example: points
;; with the attribute "external" are actually plotted when
;; CIF is created; the power attribute is used by the
;; function power-line-positions [in the MacPitts program
;; organelles.l] to find Vdd or Vss locations . These
;; positions are then used by layout-metal-lines [in
;; organelles.l] to lay down a metal line grid.
     replace-point-attributes
```

An example now shows the creation of a **point**:

```
-> (make-point '(in) 1 2 'CM
                         '((power)(external)))
;; Make a point whose name is "in", located on CMOS metal
;; at (1 2), and with "power" and "external" attributes.
     ((in) 1 2 CM ((power)(external)))
```

An item's fifth field is a summary of other items used to construct the item. This <called-symbol-names> field is composed of a list of numbers. These numbers represent **symbols**. A **symbol** is a structure containing an item's salient information. Computer time and memory use are reduced when frequently used items are constructed once and then referred to whenvever needed. Whenever an item is made using the **defsymbol** function [See Section IV.C.1], a pseudo-item, a **symbol**, is placed in the **L5-symbol-list**. Any use of this item will be reflected in the <called-symbol-names> field; these numbers indicate a **symbol**'s position in the **L5-symbol-list**. A **symbol** has the following structure:

## B. ITEMS AND THEIR OPERATIONS

The **item** data structure is the basic building block in L5. However, having to use the **make-item** function can be a bit tedious. Therefore, L5 has primitive functions for creating rectangles [or boxes] and marks [a **point** that has an **item** format]. L5 also has operators for moving, rotating, etc., **item**s and their **points**. **Item**s and **mark**s can be grouped together to form larger units using the **merge** function.

### 1. Item Creation

L5 has four functions for creating primitive items: **null-item**, **rect**, **box** and **mark**:

TABLE 4.4

FOUR PRIMITIVE ITEM CREATING FUNCTIONS

| Function | Arguments |
|---|---|
| **null-item** | none |
| **rect** | <layer><$x_{min}$><$y_{min}$><$x_{max}$><$y_{max}$> |
| **box** | <layer><length><width><$x_{center}$><$y_{center}$> |
| **mark** | <name><x><y><layer><attributes> |

Some examples of these primitive functions are:

```
-> (null-item)<CR>
;; A null item is useful as a default value for a conditional since
;; it has an item's format with only null fields [Crouch, 1983, p.5]
        (nil nil nil nil nil nil (null))

-> (rect 'CB 0 1 4 8)<CR>
;; A rectangle has no points or symbol calls. It consists of its
;; MBB coordinates (0 1) and (4 8) and a rect-tree.
      (0 1 4 8 nil nil (rect CB 0 1 4 8)[7])
```

[7] Note the difference between the <LISP form>, **(rect 'CB 0 1 4 8)**, and the <expression>, **(rect CB 0 1 4 8)**. The first is a function, the second is a data object. The first evaluates its arguments, the second is a list of parameters. Refer to Section II.C.1.

TABLE 4.5

TRANSLATION AND MERGING OPERATORS

| Function | Arguments | Description |
|---|---|---|
| **move** | <item><dx><dy> | move an item.by dx and dy units |
| **home** | <item> | place item's top left at (0 0) |
| **first-quadrant** | <item> | place item's bottom left at (0 0) |
| **second-quadrant** | <item> | place item's right bottom at (0 0) |
| **third-quadrant** | <item> | place item's right top at (0 0) |
| **fourth-quadrant** | <item> | same as home |
| **merge** | <item>$^+$ | make one item out of several items |
| **merge-list** | (<item>$^+$) | make one item out of a list of items |
| **align** | <item> <point-name> <coordinate> | move an item so that the named point is placed on the given coordinate |
| **align-items** | <item>$_1$ <point-name>$_1$ <item>$_2$ <point-name>$_2$ | <item>$_2$ is moved so that its named point aligns with <item>$_1$'s point |
| **rotcw** | <item> | rotate 90° clockwise about (0 0) |
| **rotccw** | <item> | rotate 90° counter-clockwise . . . |
| **mirrorx** | <item> | mirror about the x axis |
| **mirrory** | <item> | mirror about the y axis |

A brief look at the application of some these functions follows:

```
-> (move
        '(0 0 10 10 nil nil (rect NM 0 0 10 10)) 3 5)<CR>
;; Move the metal rectangle to the right 3 units and up 5 units.
;; Notice how only the MBB is changed [addition and consing
;; elements into a list are fast]. I.e. The result of the operation
;; could have been: (3 5 13 15 nil nil (rect NM 3 5 13 15)), but
;; if the tree was composed of many elements then each one
;; would need to be moved also!
   (3 5 13 15 nil nil (move (rect NM 0 0 10 10) 3 5)
```


Figure 4.1 **(layout-inverter 4 t)**

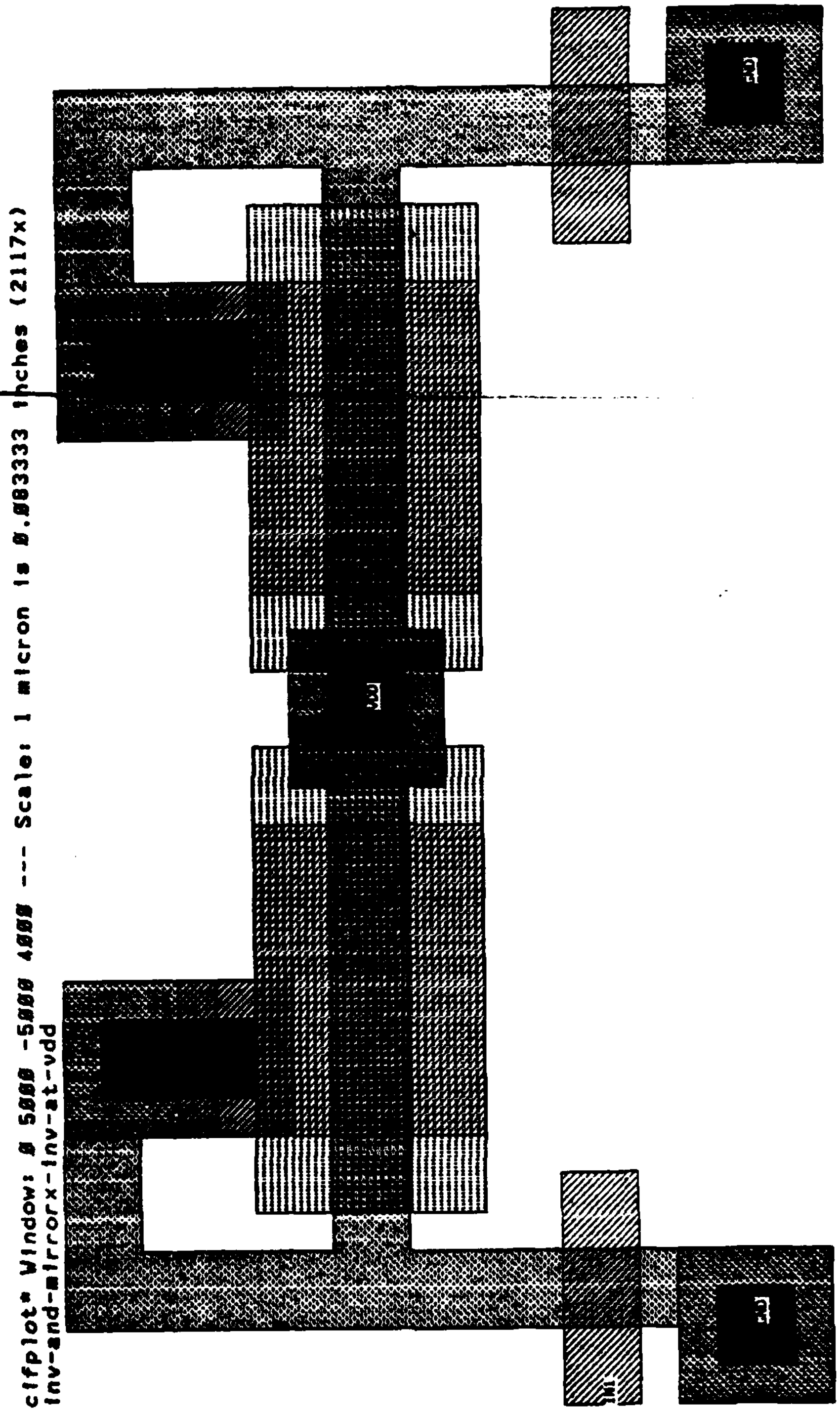



Figure 4.2 **(align-items inverter 'vdd (mirrorx inverter) 'vdd)**


Figure 4.3 **(merge inverter (move inverter 20 0))**

TABLE 4.7 (CONTINUED)

POINT OPERATORS

| Function | Arguments | Description |
|---|---|---|
| **contain** | <item><br><name> | prepend the given name to every point's name in the item |

The following examples show how point operators work. The **layout-inverter** introduced in Section IV.B.2.B is again used here. This time the +5 Volt power point is extracted from the item:

```
-> (find (layout-inverter 4 t) 'vdd)<CR>
;; Find the first point named "vdd" in a layout-inverter. Refer to
;; the previous example for (layout-inverter 4 t).
   ((vdd) 8 -2 NM (power))
```

A more complex item, **layout-and**, is shown below [Figure 4.4]:

```
-> (layout-and 4 4 t)<CR>
;; Another MacPitts organelle. This one "ands" two inputs. Note
;; that the organelle calls <symbol>s1,4,6,8,9 &10. It itself is
;; <symbol>10. The list (1 4 6 8 9 10) shows the symbols.
      (0 -43 25 0
            (((gnd) 18 -18 NM (power))
             ((vdd)  8  -2 NM (power))
             ((gnd) 21 -41 NM (power))
             ((in1)  14 -43 NP  (in))
             ((in2)  19 -43 NP  (in))
             ((vdd)   8 -25 NM (power)) )
            (1 4 6 8 9 10)
            (symbol-call 10) )
```

Since this item has more than one +5 Volt power point, they can be extracted using the following procedure:

```
-> (find-all (layout-and 4 4 t) 'vdd)<CR>
;; Find all points named "vdd" in a layout-and item.
   (((vdd) 8 -2 NM (power))((vdd) 8 -25 NM (power)))
```

```
-> (let
    ((test-item
        '(0 0 3 4
              (((in) 2 3 ND (external top))
               ((vdd) 1 1 NM (power left river))
               ((out) 3 3 NP (external signal top)) )
              (1 2 3)
              (symbol-call 3)) ) ) )
    (find-attributes test-item '(external top)) )<CR>
;; Find all points with "external" and "top" as a subset of their
;; attributes.[12] This method uses attributes to find points.
    (((in) 2 3 ND (external top))
     ((out) 3 3 NP (external signal top)) )
```

After a point has been used it is sometimes desirable to remove it from the item. There are several functions that accomplish this. Here are two examples of how to remove one point. The first method requires that the entire point be specified as follows:

```
-> (unmark
    (layout-inverter 4 t)
    '((gnd) 10 -10 NM (power)) )<CR>
;; Remove the point from the item.
    (0 -20 20 0
        (((vdd)  0  -2 NM (power))
         ((in1)  14 -20 NP  (in)) )
        (1 4 6 7)
        (symbol-call 7) )
```

The second way to delete a point is to use its name:

---

[12] The attributes could be a list of lists instead of a list. In that case when **find-attributes** is applied the attributes parameter has to be a list of lists. If the points are of the form:

**((**<name>**)** <x><y><layer>**(** **((**<attribute>****)) ))**

Then to use **find-attributes**:

**(find-attributes** <item> **'((**<attribute>$_A$**)**...**(**<attribute>$_L$**)))**

```
-> (let
    ((test-item
     '(0 0 3 4
        (((in) 2 3 NM ((external)(top)))
         ((vdd) 1 1 NM ((power)(left)(river)))
         ((out) 3 3 NP ((external)(signal)(top))) )
        (1 2 3)
        (symbol-cell 3) ) ) )
  (unmark-attributes-list test-item '((left)(top))))<CR>
;; Remove any points that have "left" or "top" as part of their
;; attributes. The river attribute refers to the river router.14
    (0 0 3 4 nil (1 2 3)(symbol-cell 3))
```

Once an item has been created, it may be desirable to give all its points a common name. By doing this, point functions that use a name as an argument to search for points will find all points with the common name.

```
-> (contain (layout-and 4 4 t) 'and-1)<CR>
;; Prepend the name "and-1" to every point's name.
    (0 -43 25 0
                (((and-1 gnd) 18 -18 NM (power))
                 ((and-1 vdd)  8  -2 NM (power))
                 ((and-1 gnd) 21 -41 NM (power))
                 ((and-1 in1)  14 -43 NP  (in))
                 ((and-1 in2)  19 -43 NP  (in))
                 ((and-1 vdd)   8 -25 NM (power)) )
                (1 4 6 8 9 10)
                (symbol-cell 10) )
```

Labels [**points** or **marks**] are useful as references to direct other functions. Notice how the next function, **layout-flags**[15], gives each of its points a " river " attribute. These labeled points can then be used by the **river** function to connect them to other items.

---

[14] See Section IV.B.2.d.

[15] This function is found in the MacPitts program flags.l.

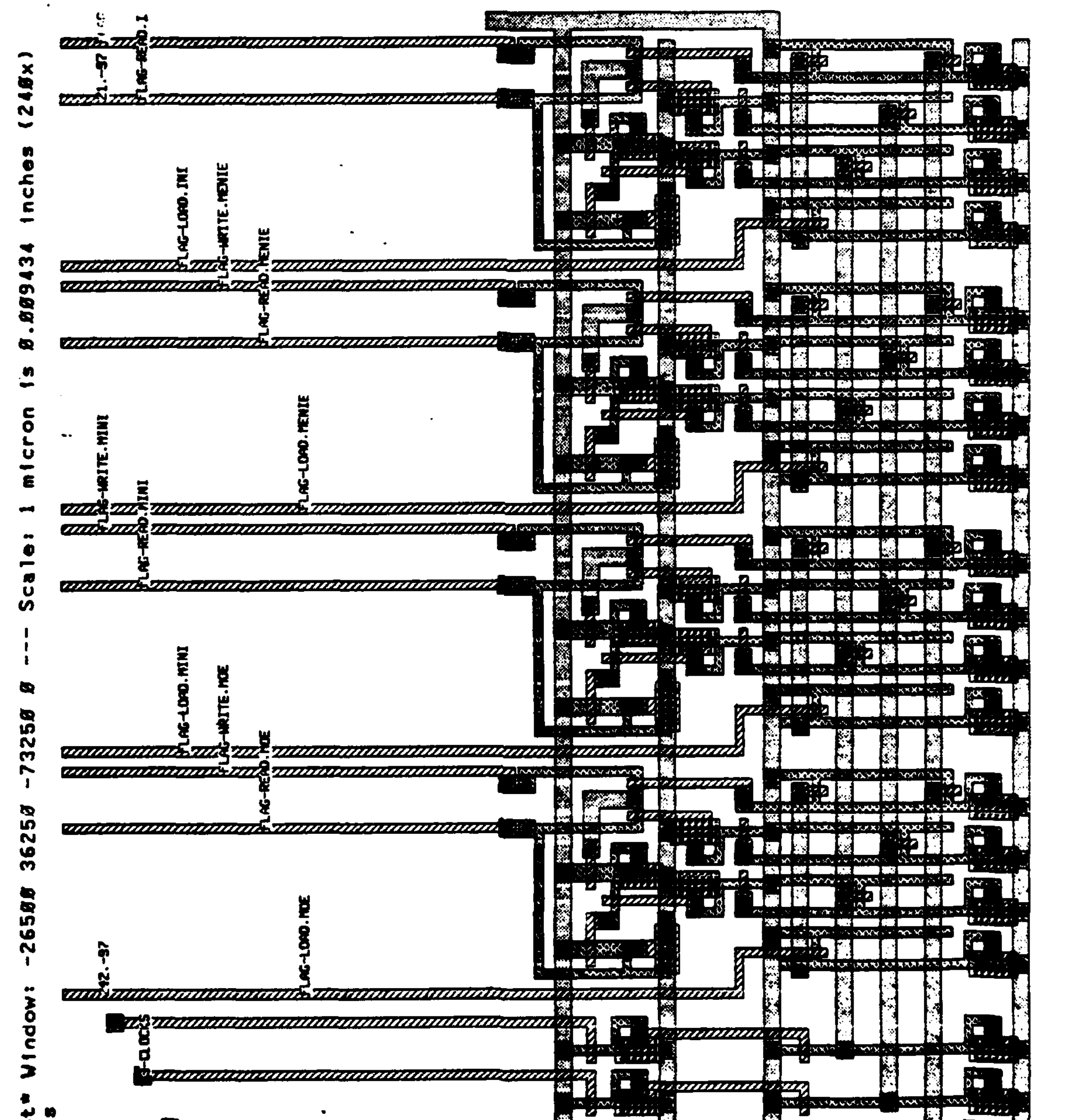



Figure 4.5 (layout-flags '(ini menie mini moe) 0 130)

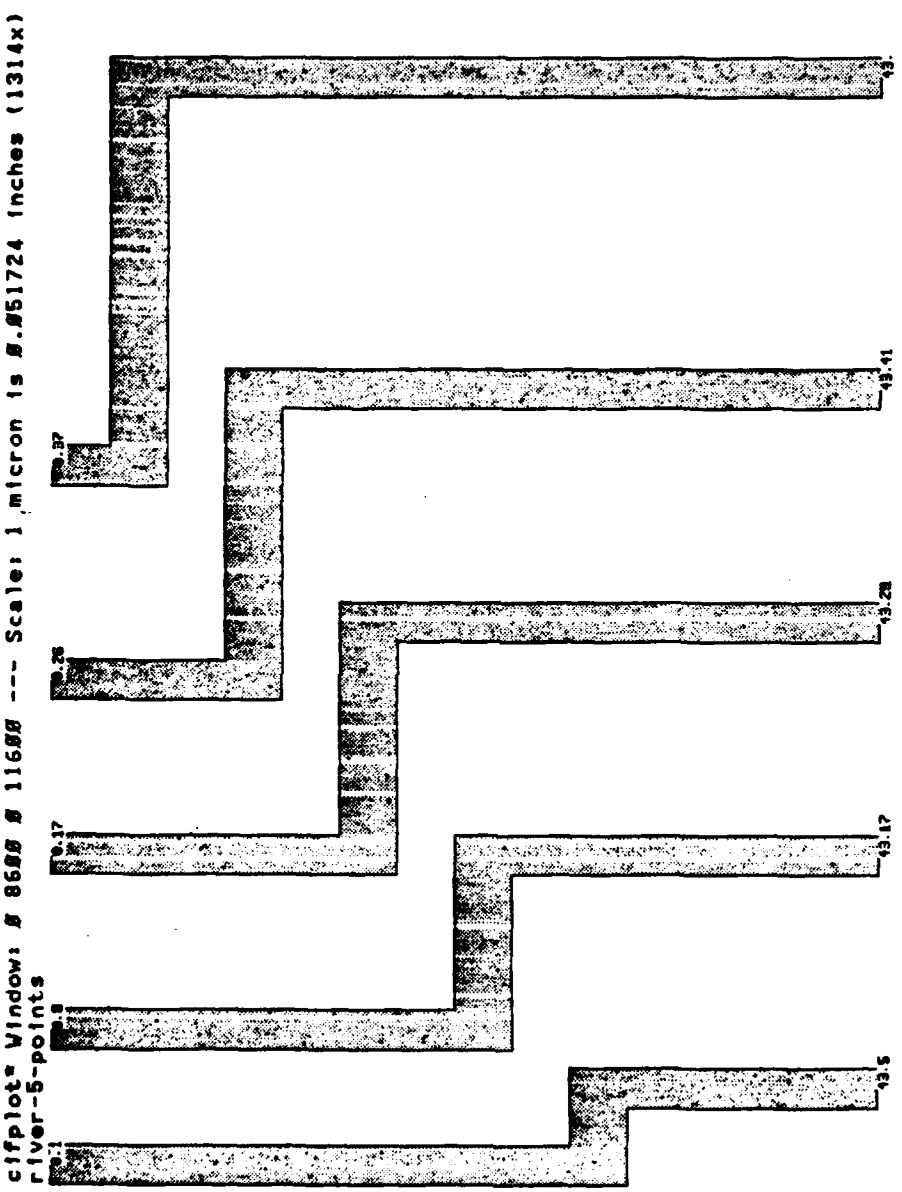



Figure 4.6 **(river 'NM 3 10 '(1 8 17 26 37) '(5 17 29 41 57))**

1. Defsymbols

In order to save memory and time, L5 has a **define symbol [defsymbol]** macro which treats items in a fashion similar to a subroutine. The **defsymbol** macro has the following syntax:

<defsymbol-name> ::=
-> **(defsymbol** <defsymbol-name>**(**<arguments>**)**<L5 form>**)**<CR>

<L5 form> ::= { <lincoln form> | <LISP form> | <L5 form> }*

When an **item** that has been defined as a **defsymbol** is called with a set of arguments it is saved as a **symbol** on the **L5-symbol-list**. Then, if it is called again by another function with the same parameters, the **L5-symbol-list** is searched for the **symbol** representing the **item**. The position of the **symbol** in the **L5-symbol-list** is returned and placed in the **called-symbol-names** field of the **item**.

If the **defsymbol** has not been called with the given set of parameters, then a **symbol** corresponding to the **defsymbol** will be placed on the **L5-symbol-list**.

There are other effects of using a **defsymbol** that depend on whether the **L5-symbol-list**'s value is **in-memory** or **on-disk**. If it's set to **in-memory** then the **item**'s **tree** is saved as part of the **symbol** that is placed on the **L5-symbol-list**. On the other hand, if it's set to **on-disk**, then the **item**'s **tree** is not stored as part of the **symbol**: the **tree** is converted to CIF and output to the **L5-symbol-file**. These two possibilities and their effects are summarized in Figure 4.7:

The storage location has been set to **in-memory**. A look at how a **defsymbol** works is now taken:

```
-> (defsymbol butting-contact ()
     (merge
        (rect 'NM 0 -6 4  0)
        (rect 'ND  0 -4 4  0)
        (rect 'NP  0 -6 4 -3)
        (rect 'NC  1 -5 3 -1) ) )<CR>
;; A butting-contact is one of L5's defsymbols.
     butting-contact
```

```
-> (butting-contact)<CR>
;; Create a butting-contact item. Notice it calls <symbol>1 in the
;; L5-symbol-list [itself].
      (0 -6 4 0 nil (1)(symbol-call 1))
```

What happened to all the layout information in the **butting-contact**? It has been placed on the **L5-symbol-list**.

```
-> (L5-symbol-list)<CR>
;; The L5-symbol-list only contains a symbol for butting-contact.
;; symbol-ID := (butting-contact 4)
;; symbol-nest-level := 1
;; symbol-tree := ((rect NM 0 -6 4 0)...(rect NC 1 -5 3 -1))
     (((butting-contact 4) 0 -6 4 0 nil nil 1
        ((rect NM 0 -6   4   0)
         (rect ND  0 -4 -4   0)
         (rect NP  0 -6   4 -3)
         (rect NC  1 -5   3 -1))))
```

A **defsymbol** can be retrieved as a function from the **L5-symbol-list** using the **L5-item-to-program** function using this syntax:

```
(def <function-name> (lambda nil <L5 form>*)) ::=
  -> (L5-item-to-program <item form><function-name>)<CR>

<item form> ::= { <item> | <defsymbol-name> }
```

```
-> (defsymbol layout-inverter (ratio1 mark?)
     (let
      ((diffusion
        (merge
         (move (diff-cut) 16 -16)
         (mark 'gnd 10 -10 'NM '(power))
         (cond
          ((= ratio1 4)(rect 'ND 7 -10 16 -16))
          (t (rect 'ND 7 -20 16 -16))))))
       (gate (rect 'NP 13 -20 15 -14))
       (mark
        (cond
         (mark? (mark 'in1 14 -20 'NP '(in)))
         (t (null-item)))))
    (merge diffusion gate mark (layout-pullup))))<CR>
;; The desymbol macro returns a name.
           layout-inverter
```

If many large items are placed on the **L5-symbol-list** and all their trees are also placed there, the list quickly becomes unwieldy. An alternative is to keep all the other information in the **L5-symbol-list**, convert the item's tree to CIF and place the CIF in the **L5-symbol-file**.

b. **on-disk** Storage

This storage mode reduces the **L5-symbol-lists** size by changing the item's tree to CIF. It is useful when items don't need to be retrieved from the **L5-symbol-list** with their trees [for example, the **L5-item-to-program** function will only create a program out of a symbol if its tree is on the **L5-symbol-list**; similary, the Caesar conversion routines (Section IV.C.3) also require **in-memory** storage].

An example can be examined after resetting all the global variables using the **start** function. First the **L5-symbol-storage** is set to **on-disk** and the effect compared with the results of Section IV.C.1.a.

2. CIF

When the **on-disk** storage mode is used, an item's tree is sent to the **L5-symbol-file** as CIF. A brief look is now taken at the CIF result from the previous example:

**-> (L5-symbol-file)**<CR>
**/tmp/L5sym20374**

**-> (exec cat /tmp/L5sym20374)**<CR>
;; exec allows UNIX® functions to be performed from LISP. In this
;; case the contents of the L5-symbol-file are concatenated to
;; the terminal. The following is CIF output. CIF uses " ( " and " ) "
;; for comments.

**DS 1;**
(define <CIF symbol> named 1);
**(name: butting-contact);**
(CIF comments are not printed out);
**L NM; B L 1000 W 1500 C 500, -750;**
(since this was output with 250 centimicrons = lambda);
(all the units must be divided by this amount);
(CIF defines its rectangles like L5 does its boxes:);
( <layer><length><width><$x_{center}$><$y_{center}$>);
( (box 'NM 4 6 2 -3) );
**L ND; B L 1000 W 1000 C 500, -500;**
( (box 'ND 4 4 2 -2) );
**L NP; B L 1000 W 750 C 500, -1125;**
( (box 'NM 4 3 2 -3) );
**L NC; B L 500 W 1000 C 500, -750;**
( (box 'NM 2 4 2 -3) );
**DF;**
(end <CIF symbol>$_1$'s definition);

The function that L5 uses to create CIF has the following format:

<file>**.cif** ::= **-> (cifout** {[']<item>}{[']<file>}{"<title>"}**)**<CR>

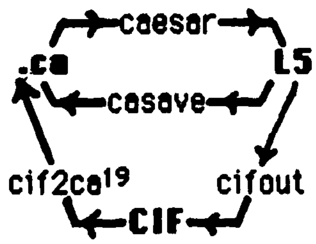



Figure 4.8 Caesar, L5 and CIF Conversions

There are several functions associated with the Caesar editor. Table 4.8 gives a synopsis (Crouch, 1983, pp. 17-18):

TABLE 4.8

CAESAR FUNCTIONS

| Function | Arguments[20] | Description |
|---|---|---|
| **caesar** | <item><file> | Converts an <item> into Caesar format, the Caesar editor is invoked and the results of the session are saved in the <file> as items. |
| **display** | <item> | Displays the <item> in Caesar without generating any L5 code afterwards. |
| **caout** | <item><file> | Outputs the <item> in Caesar format to <file>**.ca** |
| **cain** | <file> | Reads in a Caesar formatted file and converts it to L5 code. |
| **casave** | <caesar file> <L5 file> | Converts a <caesar file> into L5 code and saves the result in <L5 file>. Each Caesar symbol is made into an L5 **defsymbol**. |

---

[19] This is a Berkeley CAD conversion program from CIF to Caesar format. A minor problem with this present scheme is that Caesar evolved into the more versatile Magic system. The routines need to be modified to use Magic format instead of Caesar format.

[20] All of these functions are fexprs, therefore, none of the arguments are quoted.

# V. TOP.L AND PREPASS.L: THE TOP-LEVEL

The reader has seen how a group of LISP object files can be loaded together to create a LISP environment[1]. This chapter shows how a top-level function is used to make an environment accessible with parameters. In other words, the environment will be invoked just like other UNIX® commands. Two programs that contain top-level functions are top.l [LBS] and prepass.l [MacPitts]. In addition, these programs contain the "compilers" for LBS and MacPitts. A look at these top-level programs brings to light major differences and similarities between LBS and MacPitts.

## A. THE TOP-LEVEL

### 1. Franz Lisp's Default Top-Level

A top-level function creates the prompt-read-eval-print loop.The user can call the top-level function and can create a prompt-read-eval-print loop with different characteristics. To do this, the user defines a new top-level function and types **(reset)** to run it. (Foderado, 1983, p.13-1)(Wilensky, 1984, p.138)

When the imperative command **lisp** is given to UNIX®, the interpreter is brought into action with its default top-level: **franz-top-level**[2]. This occurs because the variable **top-level** is bound to **franz-top-level**.

---

[1] See the discussion in Chapter II Section B.2.b.

[2] Defined in /usr/lib/lisp/toplevel.l

The top-level file, top.l, is composed of several LISP functions. The first, **chip-top-level**, performs a check of the arguments used when invoking "chip". If there are no arguments then the "chip" dumplisp environment is called up. If there are arguments, then these are passed on to the **chip-compiler** function. The user should note that **chip-top-level** was set to be the top-level function in the example's makefile above.

In other words, the "chip" dumplisp environment has a function, **chip-top-level**, which handles the arguments placed in the command line when "chip" is invoked. Notice that if no arguments are given, then a message is printed out and the user is placed into the " chip " dumplisp environment. This feature can be used for debugging purposes [See Footnote 3 of this chapter]. A look at this function follows:

```
(defun chip-top-level ()
;; If "chip" is invoked without any arguments:
  (cond ((=1 (argv))
  ;; (argv) gives the number of elements on the command
  ;; line that invoked this LISP. So, if the user types:
  ;;       % chip <argument1><argument2><argument3>
  ;; then (argv) := 4
        (patom
            "usage: chip <filename> [<options>]")
        ;; (patom <expression>[<port: default to screen>])
        ;; print out the expression:
        ;;     usage: chip <filename> [<options>]
        ;; The " [ " and " ] " indicate an optional argument.
        (terpr)
        ;; (terpr) or (terpri) terminates printing.
        (setq user-top-level ())
        ;; The variable user-top-level is set to nil, but
        ;; notice that in the makefile it was set to chip
        ;; top-level. Therefore, if chip is called up
        ;; without arguments, then chip-top-level calls
        ;; up the " chip " dumplisp environment
```

```
;;      -> (count 5)
;;         (1 2 3 4 5)
(exit) )
```

The next function is set up in the makefile to handle interrupts. When an iterrupt is received it prints out "**chip-interrupt:**", the signal number and then exits the "chip" dumplisp environment. Here is the code:

```
(defun chip-interrupt-handler (signal-number)
;; This is used in the makefile as the function that
;; handles interrupts (2), floating exceptions (8),
;; alarms (14) and hang-ups (1). (Wilensky, 1984, p. 270)
  (patom " chip-interrupt: ")
  (patom signal-number)
  (terpr)
  (exit)) )
```

The previous functions allowed the user to invoke the "chip" dumplisp environment as a UNIX® command. The following function is used within the "chip" dumplisp environment to pass arguments to the **chip-compiler** function:

```
(def chip
 ;; The nlambda function format takes many arguments,
 ;; they are unevaluated and bound as a list to the
 ;; function's single parameter. For example:
 ;;     -> (chip adder cif obj)
 ;; then args := (chip adder cif obj)
  (nlambda (args)
   ;; (chip <filename> [<option>*])
   ;; <option> ::= {nostat | noobj | nocif | mag}
   ;; <default option> ::= stat obj cif
    (chip-compiler args) ) )
```

The next function coordinates other programs in order to produce the different types of output. It first uses the **process-option** function to set the global variable, **option-list**, to the options that have been input. Then

other hand, there may be other options, besides the default values, which the user can input. The following function examines the options the user has input and updates the option-list as required:

```
(defun process-option (option)
 (cond
  ((not (atom? option))
   (warning "Option not atom"))
    ;; Options must be atoms.
  ((and (> (length (explode option)) 2)
        (equal 'n (car (explode option)))
        (equal 'o (cadr (explode option))) )
   ;; Is the option more than two letters long and its
   ;; first two letters an "n" and "o" [the option is of the
   ;;  form: noXXX]? Explode separates an atom into the
   ;; characters that compose it (implode is its dual).
   (cond
    ;; Is the rest of the option [i.e. excluding the " no "]
    ;;  in the option list?
    ((member?
      (implode (cddr (explode option)))
      option-list )
     ;; Drop the option from the option-list.
     (setq option-list
           ;; Remove the indexed element from the
           ;; option-list.
          (nthdrop
           ;; Find the index of the option without the
           ;; " no " in the option-list.
           (iota
            (implode (cddr (explode option)))
            option-list )
           option-list ) ) ) ) )
   ;; Otherwise, if the option is not of the form noXXX,
   ;; then add it to the option list.
   (t (cond ((not (member? option option-list))
           (setq
            option-list
            (cons option option-list) ) ) ) ) ) )
```

expressions. This "obj" format is then used by **layout-cmos-main** [called by **layout-chip**] or **layout-inside** [used in the Caesar section] to create the array. These ideas can be seen in the implementation of LBS's compiler. The **lbs-compiler** function assumes that an **lbs-top-level, lbs-interrupt-handler** and **lbs** function are available [See Section V.A]. LBS's compiler has the following format:

```
(defun lbs-compiler (args)
 (cond
  ;; If the arguments aren't empty, then process them.
  ((not (null args))
   (mapcar 'process-option (cdr args) )
   (prog
    ;; Define local variables.
    (in-file stat-file caes-file obj-file out-file
     inport statport caesport objport bs chip )
    (setq in-file (concat (car args) '.lbs))
    (setq stat-file (concat (car args) '.stat))
    (setq caes-file (concat (car args) '.ca))
    (setq obj-file (concat (car args) '.obj))
    (setq out-file (car args))
    ;; Check that the input file is not empty and then pro-
    ;; proceed to process the input file.
    (cond
     ;; Probe the input file to see if it has anything in it.
     ;;  If it's not empty then turn it into the in port.
     ((probef in-file)
      (setq inport (infile in-file))
      (setq statport (fileopen stat-file 'w))
       ;; The boolean input format is converted to a
       ;; format showing connectivity and logical
       ;; relationships.
      (setq
         bs
         (bool-to-straps (read inport) statport) )
      ;; Check the options and produce the ones desired.
      (cond
          ;; Produce the intermediate "obj" format.
```

LBS has a simple architecture based on implementing combinational logic circuits in CMOS. MacPitts is a larger program with many more possibilities.

## C. MACPITTS[5] COMPILER

The increase in complexity from LBS to MacPitts can easily be seen in the syntax used by the latter program. A glance through MacPitt's BNF shows that it incorporates concepts such as: **function**, **macro**, **port** [n-bit data], **signal** [t or f data], **register** [datapath storage], **flag** [signal storage], **organelle** [functional unit], **test** [e.g., = or =0], etc.. MacPitts, unlike LBS, requires that I/O pads be specifically declared [that's why <pin-number>s are used to specify their location]. Again it should be noted that most of these ideas are implemented as **defstructs**.[6] Skim through the BNF to gain a feeling for MacPitts' syntax:

TABLE 5.2

MACPITTS SYNTAX

| Category | Syntax |
|---|---|
| <MacPitts program> ::= | **(program** <program-name><word-size> {<eval> \| <def> \| <always> \| <process>}+**)** |
| <eval> ::= | **(eval {compile \| simulate \| both}** <LISP form>**)** |
| <def> ::= | **(def** <pin-number> { **power \| ground \| phia \| phib \| phic**}**)** |
| <def> ::= | **(def** <register-name> **register)** |

[5] Only a brief description is given here of MacPitts. The reader should consult Southard [RVLI-3], 1983, pp. 1-33 and Siskind, 1981, pp. 1-18.

[6] This will be covered in more detail in Chapter VI.

TABLE 5.2 (CONTINUED)

MACPITTS SYNTAX

| Category | Syntax |
|---|---|
| <form> ::= | <integer> \| "<character>" \| <constant-name> \|<br><register-name> \| <port-name> \|<br>(**go** <form>) \|<br>(**call** <form>) \|<br>(**return**) \|<br>(**par** <form>$^{*}$) \|<br>(**if** <form><form><form>) \|<br>(**cond** {(<condition><form>$^{+}$)}$^{+}$) \|<br>(**setq** <register-name><form>) \|<br>(**setq** <port-name><form>) \|<br>(**setq** <signal-name><condition>) \|<br>(<function-name><formula>$^{+}$) \|<br><macro> |
| <macro> ::= | (**macro** <macro-name>{**single** \| **list**}<br><LISP form>$^{+}$) |
| <organelle> ::= | (**organelle** <organelle-name><br><#control-lines><#parameters><#test-lines><br><result?><GEN form><SIM form>) |
| <function> ::= | (**function** <function-name> <organelle-name><br>({**integer** \| **boolean**}$^{+}$)<br>(<control-line>$^{*}$)(<parameter>$^{+}$)<INT form>$^{+}$) |
| <test> ::= | (**test** <test-name><organelle-name><br>({**integer** \| **boolean**}$^{+}$)(<control-line>$^{*}$)<br>(<parameter>$^{+}$)<test-line><INT form>) |
| <condition> ::= | **t** \| **()** \| <signal-name> \| (**and** <condition>$^{+}$) \|<br>(**or** <condition>$^{+}$) \| (**not** <condition>) \|<br>(**nor** <condition>$^{+}$) \| (**nand** <condition>$^{+}$) \|<br>(**xor** <condition>$^{+}$) \| (**equ** <condition>$^{+}$) \|<br>(**bit** <bit#>{<integer>\|<integer form>}) \|<br>(**setq** <signal-name><condition>) \|<br>(<test-name><form>$^{+}$) \|<br><macro> |

TABLE 5.3

PREPASS.L FUNCTION SYNTAX

| Function | Syntax |
|---|---|
| **process**-<y> | |
| <y> ::= | { <z> \| **definitions** \| **definition** \| **control-line** \| **parameter** \| **test-line** } |
| <z> ::= | <zz>-**definition** |
| <zz> ::= | { **power** \| **ground** \| **phia** \| **phib** \| **phic** \| **register** \| **flag** \| **signal** \| **port** \| **macro** \| **constant** \| **organelle** \| **function** \| **test** } |
| **expand**-<YY> | |
| <YY> ::= | { **process** \| **macro** \| **form-list** \| **form** \| **component-list** \| **component** } |

A quick look is now taken at MacPitts' compiler. It works in a fashion similar to the LBS compiler. First, the input forms are converted to "obj" format, and then this object code is transformed into the requested options. Here is the compiler function:

```
(defun macpitts-compiler (operands)
 (prog (file-name file object obj item)
  ;; ptime gives run and garbage collection times.
  (setq initial-ptime (ptime))
  ;; The number of garbage collections that occurred.
  (setq initial-gccount $gccount$)
  ;; If the operands are null or atoms then return to the
  ;; franz-lisp top level.
  (cond ((or (not (list? operands))
             (null operands)
             (not (atom? (car operands))))
            (patom "usage: (macpitts <filename>
              [<options>])")
            (terpr)
            (return ())))
  (setq file-name (car operands))
  ;; Set the option-list to the requested and uninhibited
  ;; default settings.
```

```
  (setq obj
    (make-object
      (purge-library
        (object-definitions object) )
      (object-flags object)
      (object-data-path object)
      (object-control object)
      (object-pins object) ) )
  (setq file
    (outfile (concat file-name ".obj")) )
  (pp-form obj file)
  (close file) ) )
;; Was CIF desired?
(cond
 ((member? 'cif option-list)
  (setq item
    (catch (layout-object object) note) )
  (cond ((null item)(return ())) )
  (herald "Outputing .cif file")
  (cifout item file-name file-name) ) )
(statistic (concat "Memory used - "
           (/ (memory) 1024) "K"))
(statistic (concat
    "Compilation took "
    (quotient
      (- (car (ptime))(car initial-ptime))
      3600.0 ) " CPU minutes" ) )
(statistic (concat
   "Garbage collection took "
   (quotient
    (- (cadr (ptime))(cadr initial-ptime))
    3600.0 ) " CPU minutes" ) )
    (statistic (concat
    "For a total of "
    (- $gccount$ initial-gccount)
    " garbage collections") )
(return t) ) )
```

In summary, a bird's eye view of LBS's and MacPitts' compilers shows the relative differences between the two programs. MacPitts has a more

# VI. ORGANELLES

Chapter V contains MacPitts' syntax and its top-level function. MacPitts' BNF allows the user to define functions, macros, tests and organelles and use them when writing a MacPitts program. Alternatively, the user can modify the organelles.l and library programs and remake MacPitts. In this fashion the new operators become part of MacPitts' syntax.

## A. OVERVIEW

Before showing an example of the changes that are made in MacPitts to change its syntax, the relationships among some of its programs and functions need to be pointed out. When the user inputs a <MacPitts program>, the compiler [located in prepass.l] parses through the <MacPitts form>s. The program uses its **get**-<x>, **process**-<y> and **expand**-<yy>[1] functions to process <definition>s; evaluate <eval>s; expand <macro>s; and, obtain <source>s, <destination>s and <label>s. This is done by using list selectors to disassemble the <MacPitts program> while checking the syntactic labels that were used. For example, the words **def**, **ground**, **process**, **macro**, **function**, etc., all trigger the use of the **process-definition** function. The **eval** and **process** labels are treated separately. The functions used during the parsing process to obtain <definition>s from the input are shown below:

---

[1] Refer to Section V.C.

Prepass.l is coordinates the conversion of the <MacPitts program> to a layout with its **get-object** function. This operator uses subsidiary programs [primarily extract.l] to produce an intermediate result, an " .obj " file, which can then instantiated into the silicon mask level by layout functions [in flags.l, control.l and data-path.l]. This object file is a **defstruct** with the following definition:

**(defstruct object**
**(definitions flags data-path control pins))**

A quick look is now taken at the names of the major functional categories in prepass.l and its helper programs. Skimming through the function names provides a " feeling " for MacPitts. The most common operators are summarized by program in Table 6.1.

TABLE 6.1

MACPITTS PROGRAM FUNCTION SUMMARY

| Program | Function Name Format |
| --- | --- |
| prepass.l | **get**-<x>, **process**-<y>, **expand**-<zz> |
| extract.l | **extract**-<A> |
| | <A> ::= { **component-list** \| **process** \| **form** \| **atom** \| |
| | **list** \| **string** \| **fixnum** \| **register** \| **flag** \| **port** \| |
| | **signal** \| **label** \| **go** \| **call** \| **return** \| etc. } |
| frame.l | **layout**-<B> |
| | <B> ::= { **object** \| **skeleton** \| **wing** \| **net** \| **pins** \| |
| | **power-ring**} |
| control.l | **layout**-<C> |
| | <C> ::= { **control** \| **driver** \| **mpx** \| **register** \| |
| | **weinberger**-<D> \| etc. } |
| | <D> ::= { **gates** \| **nor** \| **nor-inport** \| **nor-gnd-line** \| |
| | etc. } |
| data-path.l | **layout**-<E> |
| | <E> ::= { **data-path** \| **buses** \| **unit** \| **organelle** \| etc. } |

creations of these data structures.[4] The functions that are implemented in the library are only constrained by the designer's imagination and a bit-slice regime.

The library is used by the **lookup-**<> function to correlate a function name found in a <MacPitts program> with an already created functional form. This can be shown in a simple example. Assume that the library is:

```
-> (setq library '((library)(constant t (nor))
              (function 1+ ...)(test = ...)))<CR>
```

Then the equality test, =, can be found as follows:

```
-> (lookup-test '= library)<CR>
;; Find the " = "test definition form in the library.
     (test = ...)
```

In summary, MacPitts relies heavily on **defstructs**. A <MacPitts program> is parsed and converted into a **defstruct** called an object. The five portions of this object are then *converted* into L5 by different programs. A particular set of layout functional units is included as a **defstruct** which contains information relating the unit to MacPitts' syntax in the library. A corresponding L5 layout of the unit is found in organelles.l. L5 in turn is a language composed of **defstructs** and layout operators.

With the general idea in mind of how MacPitts coordinates its various parts to produce a chip, consideration is next given to modifying an organelle and implementing it functionally.

---

[4] In Section VI.B an example will be traced all the way from the layout to the test definition.

## B. AN EXAMPLE

The organelle used in this example was designed by Lieutenant Anthony Mullarky using Magic. The approach used was based on (Fox, 1983, p. 32). Like Fox, a modification was made to the equality test organelle to reduce its size and increase its speed. The organelle is implemented so that its result pulls down the output bus to Vss when the test fails. Two different cells are used: $bit_0$ and a $bit_N$. The zero bit organelle is a one bit equality checker tied to Vdd in order to precharge the output bus to +5 Volts. The Nth bit organelle is a one bit equality tester without a pullup. The appellation " == " is used to differentiate this equality test from MacPitts' " = ".

The first items needed are an **organelle-==-bit-0** and **organelle-bit-==-bit-n**. The organelles were made using Magic and output as CIF. The CIF was converter to Caesar format and then into L5 format. The two organelles are shown in Figures 6.4 and 6.5.

These two organelles are then incorporated into the standard MacPitts library. Organelles.l, the compiled portion of the library, is composed of a default set of MacPitts functional units in L5 format. Adders, decrementers, equality testers, etc., are all located in organelles.l. The L5 layouts are usually defsymbols and have a name of the form: **layout-<X>-organelle**. The two basic zero and Nth bit items were made into defsymbols without any arguments.

```
(defun layout-==-organelle (ratio bit)
;; Doesn't use the ratio input.
  (cond ((=0 bit)(organelle-==-bit-0))
        (t (organelle-==-bit-n))))
```

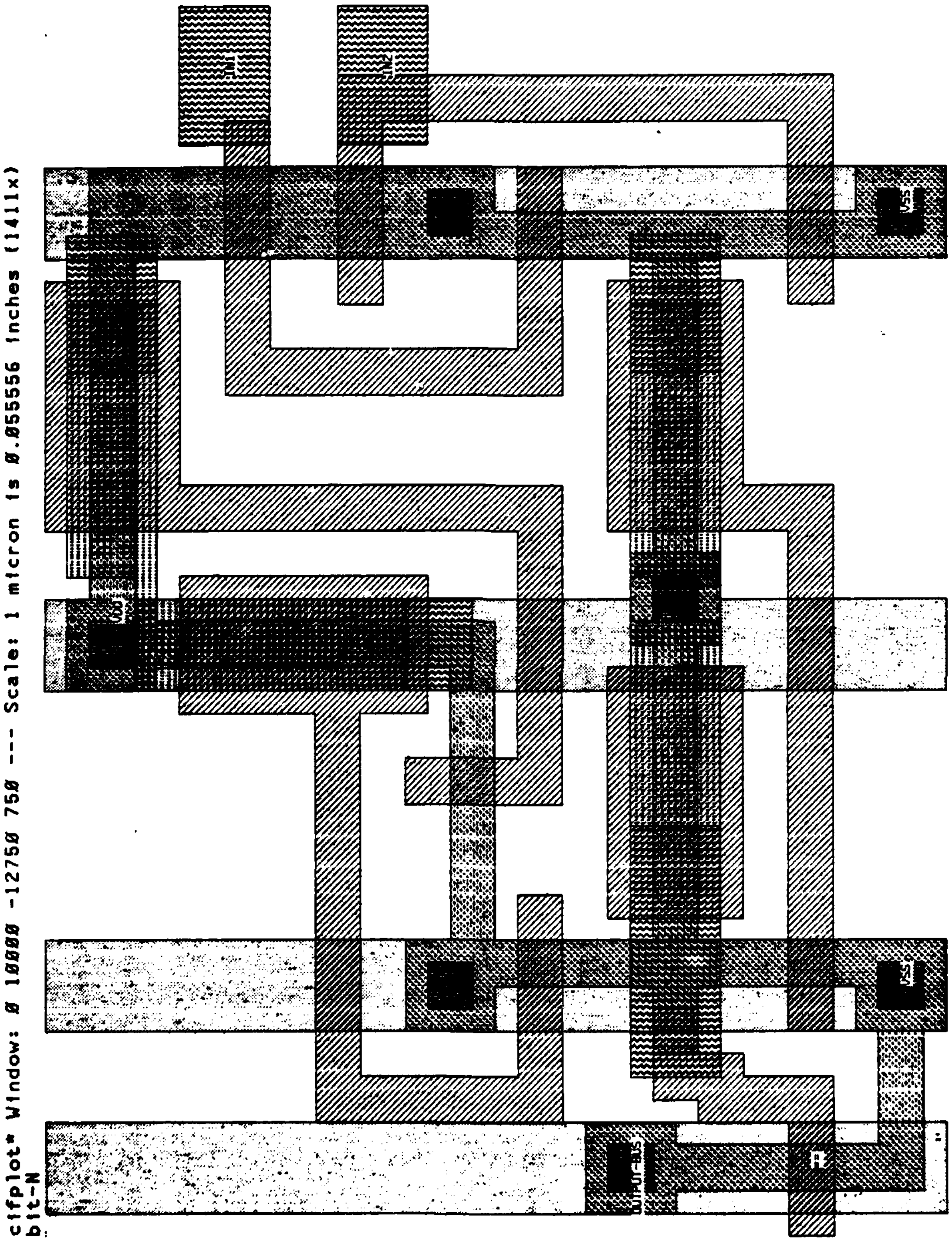



Figure 6.5 **(organelle-==-bit-n)**

Since " == " is a <test> operator, a test form is created to give the organelle functionality. This form is placed in the library along with the organelle data structure. The code for <test> " == " is:

```
(test == == (integer integer) ()
  ;; <name> := ==, <organelle> := ==,
  ;; <types> := (integer integer), <control-lines> := nil
  ;; <parameters> := ((position 1)(position 2))
  ;; <test-line> := (physical 1)
  ((position 1) (position 2)) (physical 1)
  ;; <interpret-form> follows:
  (lambda (form word-length x y)
       (cond ((or (eq x 'undefined-integer)
                  (eq y 'undefined-integer))
                  'undefined-boolean)
             ((= x y) 't)
             (t 'f))))
```

A MacPitts program is now run to check this new operator [Figure 6.6].

```
        (program five== 4
;; Example of a MACPITTS algorithm to test a 4-bit
;; integer's equality with the number five.
;; <filename> := five==.mac
 (def 11 power)
 (def 1 ground)
 (def 2 phia)
 (def 3 phib)
 (def 4 phic)
 (def in port input (5 6 7 8))
 (def out signal output 9)
;; A reset pin is needed to initialize the chip.
 (def reset signal input 10)
  (process equality 0
  first
   (cond
       ;; If " in " is " == " to 5 then set " out " to t.
       ((== in 5)(setq out t)(go first))
       ;; Otherwise, test " in " again.
       (t (go first)) ) ) )
```

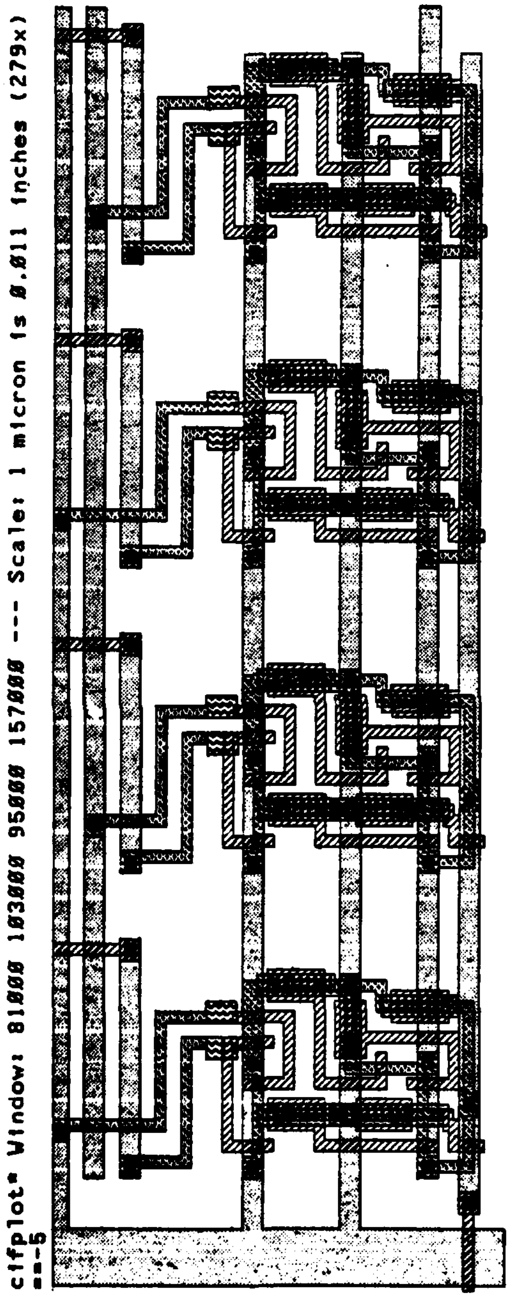



Figure 6.7 Closeup of == Organelle in five==.mac

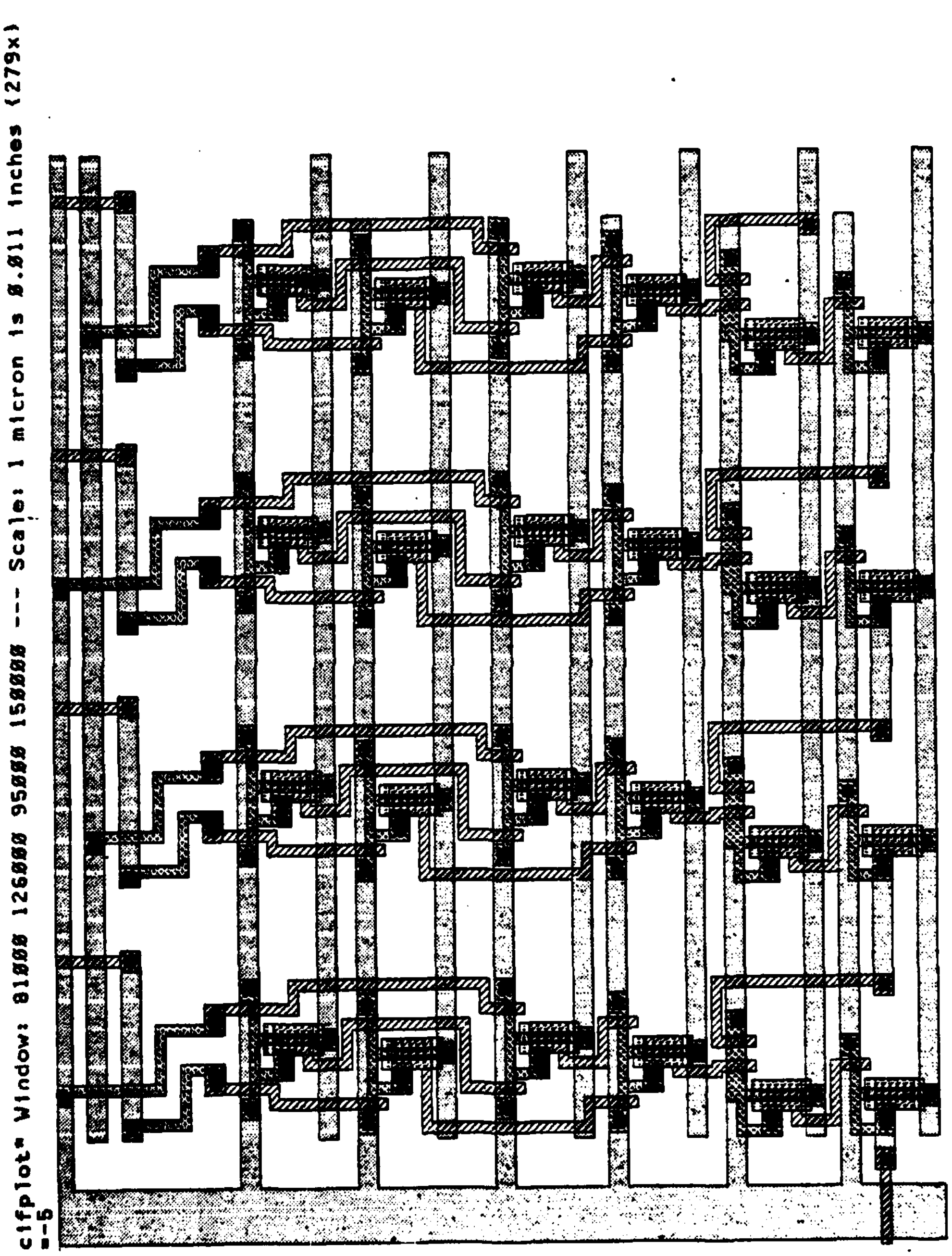



Figure 6.9 Closeup of = Organelle **five=.mac**

# VII. CONCLUSIONS

This thesis' goal was to examine L5 and show how it is used in two silicon compilers: LBS and MacPitts. The thesis showed that L5 is an extension of LISP specifically aimed at VLSI synthesis. LISP's ability to treat functions as data to create new operators was found to be the basis for the versatile **defstruct** data structure generator. The main result of the thesis is the incorporation into one document of enough information to enable a VLSI designer to automate portions of the layout process or change existing MacPitts functions to meet other needs.

For example, an examination of L5's layout primitives and data structures showed its compatibility with Caesar and CIF. However, since the graphical editor now being used at the Naval Postgraduate School is Magic, a method for using this format with L5 is needed. The suggested approach is to use the structure of L5 programs that convert Caesar into L5 and vice versa; and instead, make the conversion directly from CIF to L5. This would make available a larger pool of circuits which have been converted to CIF for incorporation into the compilers. Additionally, it would buffer the system from other changes in graphical editor file format since CIF is a widely used format.

Many possible changes to MacPitts have been recommended by Carlson, Froede and Larrabee; among these suggestions, is to allow pin locations on all four sides of the chip. In light of what has been presented in this thesis this would be a fairly straightforward alteration.

## APPENDIX A: MISCELLANEOUS TOPICS

### A. LAYOUT ERRORS

There are two types of errors associated with layouts created from Macpitts: either the code has been improperly written, or the output CIF is being plotted at the wrong scale.

An example of improper code is the erroneous specification of Vdd, Vss and output locations when a new organelle is created. In Section VI.B a new organelle, " == ", was input into MacPitts. If different parameters [shown in Table A.1] had been specified in the organelle structure, then the results of Figure A.1 would result. Notice that the interconnecting lines have all *shifted to the right* 3 λ *units.*

TABLE A.1

ORGANELLE SPECIFICATION COMPARISON

| Correct | Erroneous |
|---|---|
| **((eq info 'length) 58)** | **((eq info 'length) 52)** |
| **((eq info 'width) 40)** | **((eq info 'width) 40)** |
| **((eq info 'inputs) '(26 31))** | **((eq info 'inputs) '(26 31))** |
| **((eq info 'vdd) '(43))** | **((eq info 'vdd) '(40))** |
| **((eq info 'gnd) '(8 28))** | **((eq info 'gnd) '(5 25))** |
| **((eq info 'daisy) '(51))** | **((eq info 'daisy) '(48))** |
| **((eq info 'test) '(51))** | **((eq info 'test) '(48))** |

Additionally, the organelle length was specified to be several λ units longer than the layout actually is, to prevent the output line (when routed to the Weinberger array) from shorting with the clock lines [See Figure A.2].

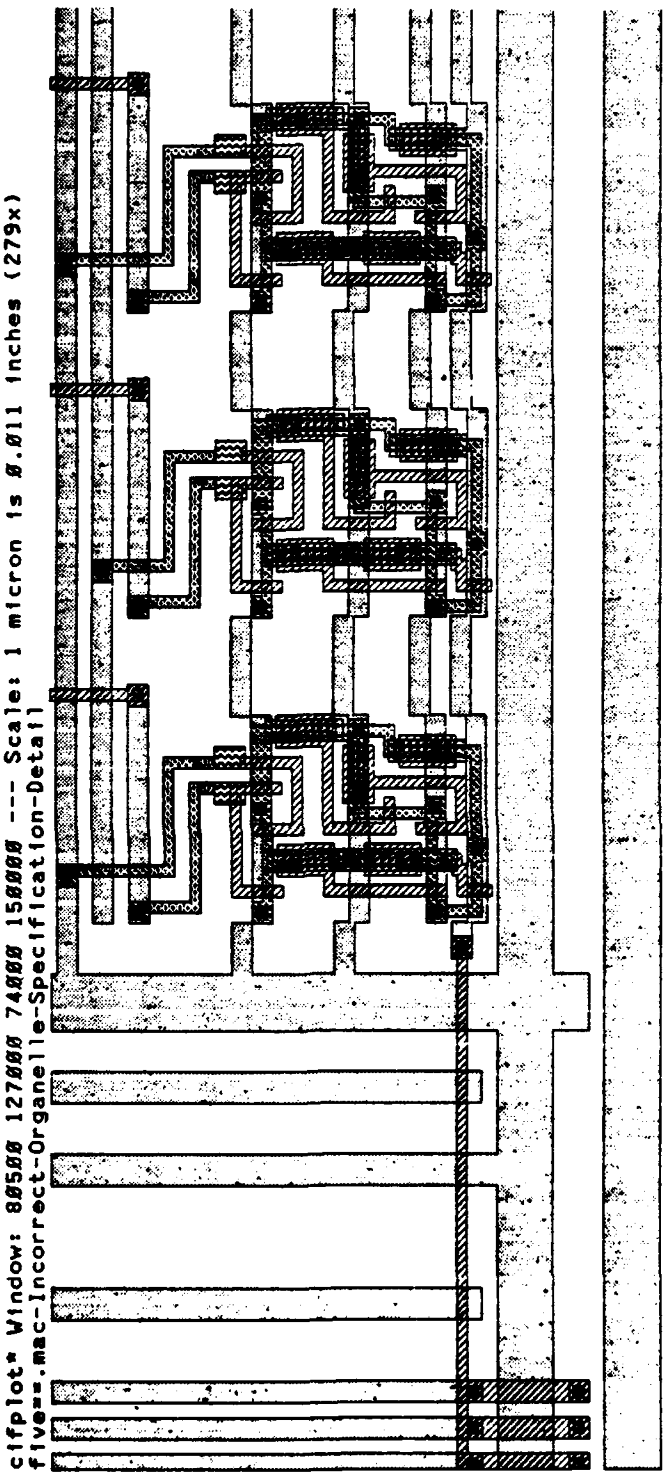



Figure A.2 **five==.mac** Organelle Detail

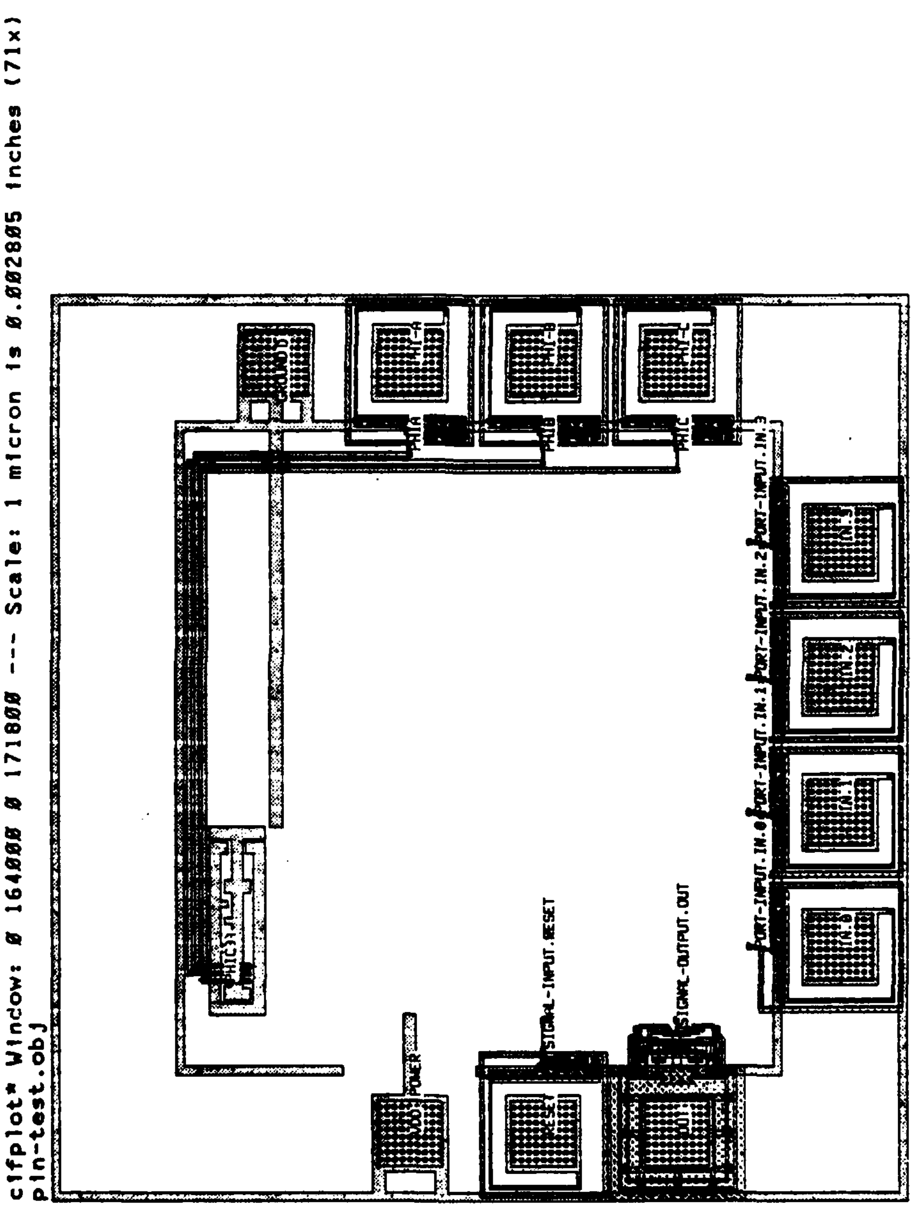



Figure A.3 **five=.mac** Pins With Correct CIF Scale

The user now has all of MacPitts available with the exception of the organelles and library.

Instead of running MacPitts every time a change is made, the intermediate object code will be modified. Object code is generated when five==.mac is run in the following manner:

```
% macpitts obj cif noint noopt-c noopt-d
nostat five==.mac & > trash
*Create an object and CIF file. Redirect comments
*to a trash file.
```

Now, assuming that the macpitts environment has been invoked as shown above [**% macpitts**], two local files that contain commands to change UNIX® directories and plot L5 items are loaded.

```
-> (include edit.l)
 [load edit.l]
t

-> (include plot.l)
[load plot.l]
t

-> (minimum-feature-size! 200)
200
```

The object file that was generated by MacPitts, five==.obj, is altered by setting its data-path and controller to **nil**. After editing is complete, the new file is called pin-test.obj and is shown below:

```
-> (exec cat pin-test.obj)
;; The first portion of the object is definitions
(((source reset)
(register sequencer-equality-state)
(source sequencer-equality-state)
(destination sequencer-equality-state)
```

```
-> (thesis-plot (layout-object
    (read (infile 'pin-test.obj))) 'pin-test.obj t)
;; The standard statistics are output, except the control
;;  unit and data path are empty.
Statistic - Control has 0 columns
Statistic - Circuit has 78 transistors
Statistic - Control has 0 tracks
Statistic - Power consumption is 0.034114
      Watts
Statistic - Data-path internal bus uses 0 tracks
Statistic - Dimensions are 1.640000 mm by
      1.718000 mm
;; The rest of the output is related to the plotting
;; function.
...
-> Window: 0 164000 0 171800
Scale: 1 micron is 0.002805 inches (71x)
The plot will be 0.38 feet
```

This plot is shown in Figure A.3. The object file is modified again to place the pins in different locations:

```
-> (exec cat pin-test-2.obj)
;; A random pin ordering was chosen.
(((source reset)
  (register sequencer-equality-state)
  (source sequencer-equality-state)
  (destination sequencer-equality-state)
  (port sequencer-equality-next-state internal
                                              nil)
  (source sequencer-equality-next-state)
  (destination sequencer-equality-next-state)
  (label first equality 0)
  (destination out)
  (source first)
  (source in)
  (logo five=)
  (word-length 4)
  (power 11)
  (ground 1)
  (phin 2)
```

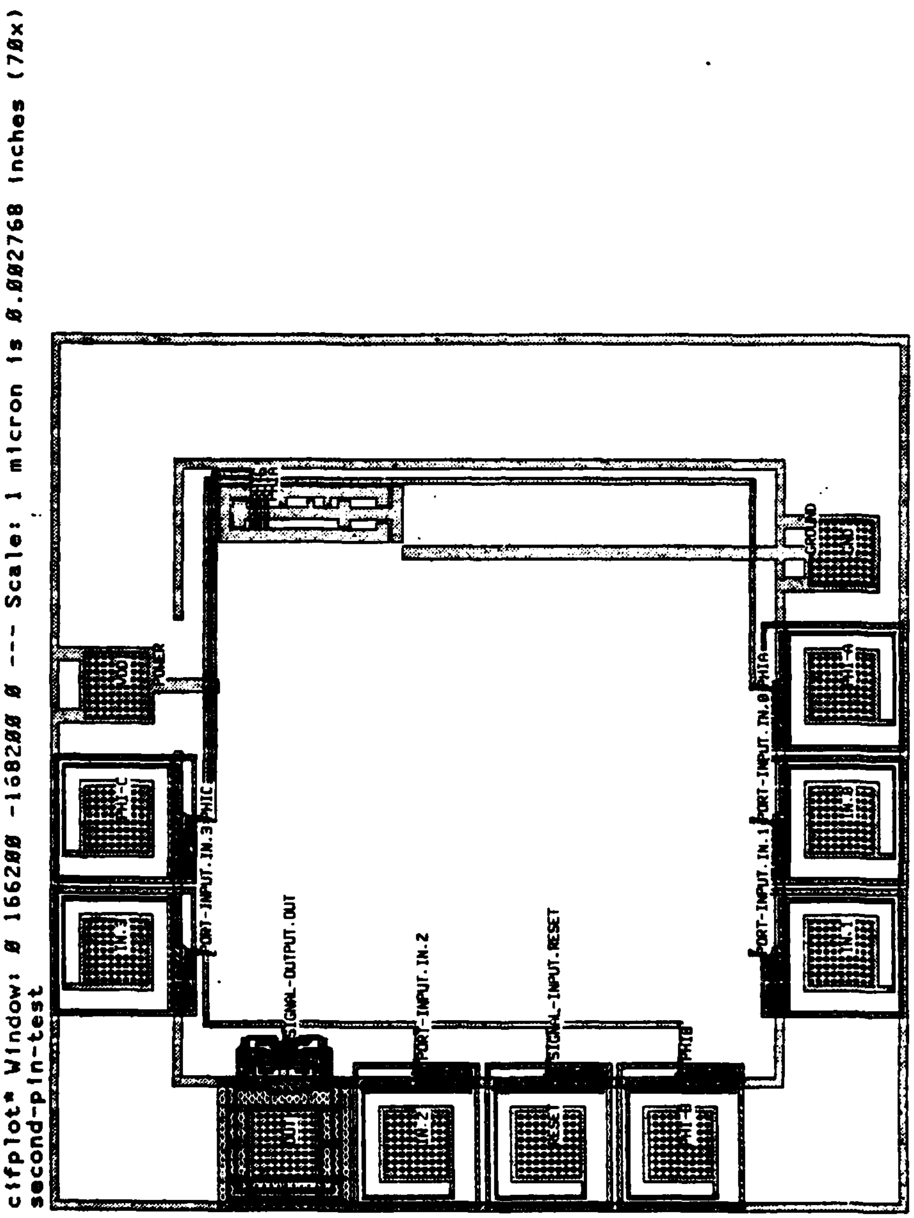



Figure A.5 **five=.mac** Modified Pins.

Hasemer, T., Looking at LISP, Addison-Wesley, 1984.

Haugeland, J., Artificial Intelligence: The Very Idea, MIT Press, 1985.

Haugeland., J., editor, Mind Design: Philosophy-Psychology-Artificial Intelligence, MIT Press, 3rd ed., 1985.

Hofstadter, D.R, Metamagical Themas, Basic Books, 1985.

Hon, R.W., and Sequin C.H., A Guide to LSI Implementation, Xerox Palo Alto Research Center, 1980.

International Business Machines (IBM) Technical Newsletter GC26-3847-5, APL Language, 6th ed., IBM Corporation, 1983.

Larrabee, R. C.., VLSI Design with the MacPitts Silicon Compiler, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

MacLennan, B. J., Principles of Programming Languages: Design Evaluation and Implementation, Holt, Rinehart and Winston, 1983.

Mead, C. and Conway, L., 2d ed., Introduction to VLSI Systems, Addison-Wesley, 1980.

Ousterhout, J.K., Editing VLSI Circuits with Caesar, Computer Science Division, Electrical and Computer Sciences, University of California, 1985.

Rosenberg, J.M., Dictionary of Computers, Data Processing and Telecommunications, John Wiley and Sons, 1984.

Scott, W.S., Hamachi, G., Mayo, R.N. and Ousterhout, editors, J.K., 1985 VLSI Tools: More Works by the Original Artists, Computer Science Division EECS Department, University of California at Berkeley, 1985.

Scott, W.S., Hamachi, G., Mayo, R.N. and Ousterhout, editors, J.K., 1986 VLSI Tools: Still More Works by the Original Artists, Computer Science Division EECS Department, University of California at Berkeley, 1986.

## BIBLIOGRAPHY


ACM IEEE Design Automation Conference Proceedings, 19th, IEEE Press, 1982.

Agre, P. E., "Designing A High-Level Silicon Compiler", Proceedings IEEE International Conference on Computer Design: VLSI in Computers (ICCD 83), November 1983.

Allen, G.H., Denyer, P.B., Renshaw, D., "A Bit Serial Linear Array DFT", Proceedings of the IEEE Conference on Acoustics, Speech and Signal Processing (ICASSP 84), March 1984.

Allerton, D.J., Batt, D.A., and Currie, A.J., "A VLSI Design Language Incorporating Self-Timed Concurrent Processes", IEE European Conference on Electronic Design Automation (EDA 84), March 1984.

Anceau, F., "CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits Specified by Algorithms", in Bryant, 1983, pp. 15-31.

Anceau, F., Aas, E.J. (editors), 'VLSI 83: Proceedings of the IFIP TC WG 10.5 International Conference on Very Large Scale Integration, North-Holland, 1983.

Anceau, F., and Schoellkopf, J.P., "CAPRI: A Silicon Compiler for VLSI Circuits Specified by Algorithms", in Randell, 1983, pp. 149-54.

Ayres, R., "Silicon Compilation-A Hierarchical Use of PLA's", Proceedings of the 16th Design Automation Conference, 1979.

Bairstow, J.N., "Chip Design Made Easy", High Technology, Volume 5, Number 6, June 1985.

Baker, A., "Selecting a Silicon Compiler", VLSI Systems Design, Volume VII, Number 5, May 1986.

Baker, S., "Silicon Compilers Puts Systems Houses in VLSI Business", Electronic Engineering Times, No. 300, 8 October 1984.

Bloom, M., "Silicon Compilation: The Fast Track to Masks", Electronic Engineering Times, Number 294, 13 August 1984.

Bond, J., "Circuit Density and Speed Boost Tomorrow's Hardware", Computer Design, Volume 23, Number 10, September 1984.

Bond, J., "Future Hardware", Computer Design, Volume 23, Number 10, September 1984.

Brayton, R.K., Chen, C.L., McMullen, C.T., Otten, R.H., and Yamour, Y.J., "Automated Implementation of Switching Functions as Dynamic CMOS Circuits", Proceedings of the 1984 Custom Integrated Circuits Conference, May 1984.

Brayton, R.K., Hachtel, G.D., McMullen, C.T., and Sangiovanni-Vincentelli, A.L., Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, 1984.

Brown, C. "Silicon Compilers: How Some Labs are Trying to Implement Them", Electronic Engineering Times, No. 300, 8 October 1984.

Bryant, R., editor, Third Caltech Conference on Very Large Scale Integration, Computer Science Press, 1983.

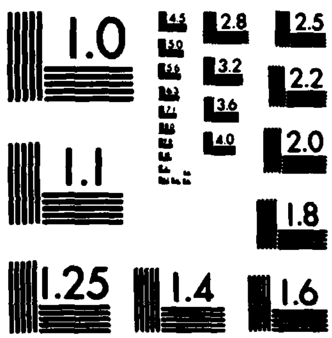
Buric, M.R., Christensen, C., and Matheson, T.G., "The Plex Project: VLSI Lay outs of Microcomputers Generated by a Computer Program", IEEE International Conference on Computer-Aided Design (ICCAD-83), September1983.

Bursky, D., "Circuit Compiler Cuts Chip Area 25 Percent to 40 Percent Over Standard Cells", Electronic Design, Volume 32, Number 18, 6 September 1984.

Carnegie-Mellon University Departments of Computer Science and Electrical Engineering Technical Report, by Barbacci, M. R., Barnes, G., Cattell, R. and Siewiorek, D., The ISPS Computer Description Language, 16 August 1979.

Carnegie-Mellon University Departments of Computer Science and Electrical Engineering Technical Report, by Barbacci, M. R., et. al. , The Symbolic

AD-A171 369 SILICON COMPILATION USING A LISP-BASED LAYOUT LANGUAGE 2/2
(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA
M A MALAGON-FAJAR JUN 86
UNCLASSIFIED F/G 9/2 NL




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Manipulation of Computer Descriptions: The ISPS Computer Description Language, March 1978.

Capello, P. R., et. al. editors, VLSI Signal Processing, IEEE Press, 1984.

Carter, T.M., Davis, A., Hayes, A.B., Lindstrom, G., Klass, D., Maloney, M.P., Nelson, B.E., Organick, E.I., and Smith, K.F., "Transforming an ADA Program Unit to Silicon and Testing it in an ADA Environment", Digest of Papers COMPCON Spring 84. Twenty-Eighth IEEE Computer Society International Conference, March 1984.

Cham, K.M., Oh, S.-H., Chin, D., and Moll J.L., Computer Aided Design and VLSI Device Development, Kluwer Academic Publishers, 1985.

Cheng, E.K., "The Design of an Ethernet Data Link Controller Chip", Digest of Papers Spring COMPCON 83. Intellectual Leverage for the Information Society, March 1983.

Clary, J.B., Denyer, P.B. (editors), "The Impact of VLSI on Digital System Design", Systems on Silicon. First IEE International Specialist Seminar 2, Peter Peregrinus, 1984.

Cline, K., Cutler, M., Kesselman, C., and York, G., "Automated Attribute Optimization for VLSI Systems", 1984 Conference Proceedings- 3rd Annual Intenational Phoenix Conference on Computers and Communications, IEEE Press, 1984.

Collet, R., "Silicon Compilation: A Revolution in VLSI Design", Digital Design, Volume 14, Number 8, August 1984.

Conference Record-16th Asilomar Conference on Circuits Systems & Computers, IEEE Press, 1982.

Cory, W.E., and van Cleemput, W.M., "Developments in Verification of Design Correctness", Proceedings 17th Design Automation Conference, 1980.

Curtis, W., "Silicon Compilation Speeds Design of Complex Chips", Computer Design, March 1985.

Curtis, W., "Designing the Micro-VAX Using Silicon Compilation", COMPCON '85 Computer Conference, IEEE Computer Society, 1985.

Cuykendall, R., Domic, A., Joyner, W.H., Johnson, S.C., Kelem, S., McBride, D., Mostow, J., Savage, J.E., and Saucier, G., "Design Synthesis and Measurement", in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 6-9; and, Journal of Systems and Software, Volume 4, Number 1, April 1984.

Darringer, J. A. and Joyner, W. H., Jr., A "New Look at Logic Synthesis", Proceedings of the 17th Design Automation Conference, 1980.

Darringer, J. A. et. al., "Logic Synthesis Through Local Transformations", IBM Journal of Research and Development, Volume 25, Number 4, 1981.

Dasgupta, S., "Computer Design and Description Languages", Advances in Computers, Volume 21, Academic Press, 1982.

DeMan, H. et. al., "Custom Design of Hardware Digital Filters on I.C's", Proceedings of the 1982 Custom Integrated Circuits Conference, 1982.

DeMan, H., Reynders. L, Bartholomeus, M., Cornelissen, J., "PLASCO: A Silicon Compiler for NMOS and CMOS PLA's", in Anceau, 1983, pp 171-81.

Denyer, P. B., Renshaw, D., and Bergmabb, N., "A Silicon Compiler for VLSI Signal Processors", Proceeding of the 8th European Solid-State Circuits Conference (ESSCIRC 82), 1982.

Denyer, P.B., and Renshaw, D., "Case Studies in VLSI Signal Processing Using a Silicon Compiler", Proceedings ICASSP, April 1983.

Denyer, P.B., Murray, A.F., and Renshaw, D., "FIRST: Prospect and Retrospect", in Capello, 1984, pp. 252-264.

Denyer, P. and Renshaw, D., VLSI Signal Processing, A Bit-Serial Approach, Addison-Wesley, 1985.

Digest of Papers- COMPCON Spring 82: High Technology in the Information Industry, IEEE Computer Society, 1982.

Frantz, D., and Rammig, F.J., "The Impact of an Advanced CHDL in VLSI Design", Proceedings International Conference on Computer Design (ICCD-83), October 1983.

"Frustration Lead Alles and Rip into the Custom-IC Business", Electronics, Volume 57, Number 13, 28 June 1984.

Fujita, T., and Goto, S., "A Rule Based Routing System", ICCD-83.

Furlow, B., "Silicon Compilation Cuts Costs of Custom VLSI", Computer Design, Volume 23, Number 12, 15 October 1984.

Gajski, D.D., "The Structure of A Silicon Compiler", IEEE International Conference on Circuits and Computers 1982 (ICCC 82), IEEE Press, 1982.

Gajski, D.D. and Kuhn, R.H.," Guest Editors' Introduction: New VLSI Tools", Computer, Volume 16, Number 12, 1983.

Gajski, D.D., "Silicon Compilers and Expert systems for VLSI", Proceedings '84 ACM IEEE 21st Design Automation Conference, June 1984.

Gazsi, L., "Explicit Formulas for Lattice Wave Digital Filters", IEEE Transactions on Circuits and Systems, Volume CAS-32, Number 1, January 1985.

Glasser, L. A. and Dobberpuhl, D. W., The Design and Analysis of VLSI Circuits, Addison-Wesley, 1985.

"Graphics Systems Chip Designed to Replace 80 TTL Components", Electronic Engineering Times, Number 284, 23 April 1984.

Gray, J.P., editor, VLSI 81. Very Large Scale Integration. Proceedings of the First International Conference on Very Large Scale Integration, Academic Press, 1981.

Gray, J.P., Buchanan, I. and Robertson, P.S., "Controlling VLSI Complexity Using a High-Level Language for Design Description", Proceedings IEEE International Conference on Computer Design (ICCD-83), October 1983.

Hilfinger, P., "Silage: A High Level-Language and Silicon Compiler for Digital Signal Processing", Proceedings CICC 1985, 1985.

Hirschhorn, S., and Davis, A.M., "The Revolution in VLSI Design: Parallels Between Software and VLSI Engineering", in Conference Record-16th Asilomar Conference on Circuits Systems & Computers, 1982; and in Workshop Report: VLSI and Software Engineering, 1984, pp. 75-84.

Hodges, D.A., and Jackson, H.G., Analysis and Design of Digital Integrated Circuits, McGraw Hill, 1983.

Holland, J. H., Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, The University of Michigan Press, 1975.

IEEE 1984 Workshop on the Engineering of VLSI and Software, IEEE Press, 1984.

Ishii, J., Sugiura, Y., and Sueshiro, Y., "A Gate Array CAD System and Future Tasks in the Field", Third Symposium on VLSI Technology, September 1983.

Jespers, P., Sequin, C., and van de Wiele, F., Design Methodologies for VLSI Circuits, Sitjthoff and Noordhoff, 1982.

Johannsen, D.L., "Bristle Blocks- A Silicon Compiler", Proceedings of the 16th Design Automation Conference, 1979.

Johannsen, D.L., Silicon Compilation, PhD. Dissertation, Technical Report 4530, California Institute of Technology, 1981.

Johnson, S.C., "Code Generation for Silicon", Proceedings 10th ACM Symposium on Principles of Programming Languages, 1983.

Johnson, S.C., "VLSI Circuit Design Reaches the Level of Architectural Description", Electronics, Volume 57, Number 9, 3 May 1984.

Johnson, S.C., and Mazor, S., "Silicon Compiler Lets System Makers Design Their Own VLSI Chips", Electronic Design, Volume 32, Number 20, 4 October 1984.

Lee, B., Ritzman, D., and Snapp, W., "Silicon Compiler Teams with VLSI Workstation to Customize CMOS ICs", Electronic Design, Volume 32, Number 23, 15 November 1984.

Lee, C.H., and Lin-Hendel, C.G., "Current Status and Future Projection of CMOS Technology", Proceedings IEEE COMPCON, Fall 1982.

Leighton, F.T., and Leiserson, C.E., "Wafer-scale Integration of Systolic Arrays", Proceeding 23rd IEEE Symposium of Foundations of Computer Science.

Leinwand, S., and Lamdan, T., "Design Verification Based on Functional Abstraction", Proceedings 16th Design Automation Conference, 1979.

Leiserson, C.E., Rose, F.M., and Saxe, J.D., "Optimizing Synchronous Circuits by Retiming", in Bryant, 1983, pp 87-116.

Liesenberg, H.K.E., and Kinniment, D.J., "Autolayout System for a Hierarchical I.C. Design Environment", Integrated VLSI, Volume 1, Numbers 2-3, October 1983.

Lopez, A.D., and Law, H.F., "A Dense Gate Matrix Layout Style for MOS LSI", Digest of Technical Papers, ISSCC 1980.

Louie, G., Ho, T., and Cheng, E., "The Microvax I Data-Path Chip", VLSI Design, Volume 4, Number 8, December 1983.

Lowe, L., "VLSI Design Shrinks to Mere Man-Weeks", Electronics, 1982.

Lukuhay, J. and Kubitz, W.J., "A Layout Synthesis System for nMOS Gate-Cells", Proceedings of the 19th Design Automation Conference, 1982.

Lyon, R.F., A Bit-Serial VLSI Architectural Methodology for Signal Processing, in Gray, 1981, pp. 131-140.

Lyon, R.F., MSSP: A Bit-Serial Multiprocessor for Signal Processing, in Capello, 1984, pp. 64-75.

Naval Postgraduate School Technical Report NPS52-81-009, The Structural Analysis of Programming Languages, by B.J. MacLennan, 9 June 1981.

1984; also in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 117-125.

Moulton, A., "Laying the Power and Ground Wires on a VLSI Chip", ACM IEEE 20th Design Automation Conference, 1983.

Murray, A.F., Denyer, P.B., and Renshaw, D., "Self-Testing in Bit Serial VLSI Parts: High coverage at Low Cost", Proceedings IEEE International Test Conference, October 1983.

Nash, J.H., and Smith, S.G., "A Front End Graphics Interface to the FIRST Silicon Compiler", IEE European Conference on Electronic Design Automation, March 1984.

NewKirk, J. A. and Matthews, R., The VLSI Designer's Library, Addison-Wesley, 1983.

Offen, R.J., VLSI Image Processing, McGraw Hill, 1985.

Organick, E.I., Lindstrom, G., Smith, D.K., Subrahmanyan, P.A., and Carter, T., Transformation of ADA Programs into Silicon, Semiannual Technical Report UTEC 82-020, University of Utah, 1982.

Ostreicher, D., "Where is Computer-Aided Design Going?", Computer, Volume 16. Number 5, May 1983.

Panasuk, C., "Silicon Compilers Make Sweeping Changes in the VLSI Design Worlds", Electronic Design, Volume 32, Number 19, 20 September 1984.

Parker, A.C. et. al., "The CMU Design Automation System: An Example of Automated Data-Path Design", Proceedings of the 16th Design Automation Conference, 1979.

Pearl, J., Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1984.

Percival, R., and Fitchett, M., "Designing a Laser-Personalized Gate Array", VLSI Design, Volume 5, Number 2, February 1984.

Reekie, H.M., Mavor, J., Petrie, N., and Denyer, P.B., "An Automated Design Procedure for Frequency Selective Wave Filters", 1983 IEEE International Symposium on Circuits and Systems, Volume 1, IEEE Press, 1983.

Reekie, H.M., Petrie, N., Mavor, J., Denver, P.B., and Lau C.H., "Design and Implementation of Digital Wave Filters Using Universal Adaptor Structures", IEE Proceedings 1984, Part F, Volume 131, Number 6, 1984.

Reiss, S.P. and Savage, J.E., "SLAP- A Silicon Layout Program", Proceedings IEEE International Conference on Circuits and Computers, 1982.

Reutz, P.A., Pope, S.P. Solberg, B., and Broderson R.W., "Computer Generation of Digital Filter Banks", Digest of Technical Papers, IEEE International Solid State Circuits Conference (ISSCC-84), February 1984.

Rivest, R.L., "The PI (Placement and Interconnect) System", Proceedings 20th Design Automation Conference, 1982.

Rosenberg, A.L., "References to the Literature on VLSI Algorithmics and Related Mathematical and Practical Issues", Sigact News, Volume 16, Number 3, Fall 1984.

Rosenberg, J.B., "Chip Assembly Techniques for Custom IC Design in a Symbolic Virtual Grid Environment", Proceeding MIT Conference on Advanced Research in VLSI, January 1984.

Rosenberg, J.B., and Weste, N.H., ABCD- A Better Circuit Description, MCNC Technical Report 4983-01, Microelectronic Center of North Carolina, 1983.

Rupp, C.A., "Components of a Silicon Compiler System", in Gray, 1981, pp. 227-236.

Saucier, G., and Serrero, G., "Intelligent Assistance for Top Down Design of VLSI Circuits", in Workshop Report: VLSI and Software Engineering Workshop, 1984, pp. 107-111.

Savage, J.E., "Three VLSI Compilation Techniques: PLAs, Weinberger Arrays, and SLAP, (A New Silicon Layout Program)", ALgorithmically-Specialized Computers, 1983.

Srini, V.P., "Test Generation for MacPitts Designs", Proceedings IEEE International Conference on Computer Design, (ICCD-83), 1983.

Stefik, M. et. al., "The Partitioning of Concerns in Digital System Design", MIT Conference on Advanced Research in VLSI, 1982.

Subrahmanyan, P.A., "Synthesizing VLSI Circuits from Behavioral Specifications: A Very High Level Silicon Compiler and its Theoretical Basis", in Anceau, 1983, pp. 195-210.

Suzim, A.A., "Data Processing Section for Microprocessor-Like Integrated Circuits", IEEE Journal of Solid State Circuits Conference, 1981.

Swartzlander, E.E., VLSI Signal Processing Systems, Kluwer Academic Publishers, 1985.

"The Evolution of Chip Customization", High Technology, Volume 2, Number 1, January 1983.

Touretzky, D. S., LISP: a Gentle Introduction to Symbolic Computation, Harper & Row, 1984.

Travassos, R.H., "Hardware Design Automation", Proceedings of the 1983 American Control Conference, September 1983.

Trickey, H.W., "Good Layouts for Pattern Recognizers", IEEE Transactions on Computing, Volume C-31, Number 6, June 1982.

Trickey, H.W. and Ullman, J.D., "Regular Expression Compiler", Digest of Papers- COMPCON Spring 82: High Technology in the Information Industry, IEEE Computer Society, 1982.

Turner, L.E., Denyer, P.B., and Renshw, D., "A Bit Serial LDI Recursive Digital Filter", Proceedings of the IEEE Internation Conference on Acoustics, Speech and Signal Processing (ICASSP 84), March 1984.

Ullman, J. D., Computational Aspects of VLSI, Computer Science Press, 1984.

VanCleemput, W.S., "Computer Hardware Description Languages and Their Applications", Proceedings 16th Design Automation Conference, 1979.

Wolf, W., Newkirk, J., Mathews, R., and Dutton, R., "Dumbo: A Schematic-to-Layout Compiler", in Bryant, 1983, pp. 379-391.

Workshop Report. VLSI and Software Engineering Workshop, IEEE Computer Society Press, 1984.

Young, J., "Why Silicon Compilers had to Change its Strategy", Electronics, 1985.

Zarrella, J., "How Silicon Compilation will Affect Engineers", Microcomputer Applications, COMPCON Spring '85 Computer Conference, IEEE Computer Society, 1985.

6. Prof. R. McGhee, Code 52Mz 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943-5000

9. Prof. D. Bukofzer, Code 62BH 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5000

7. Mr. P. Blankenship 1
Massachusetts Institute of Technology
Lincoln Laboratory
P.O. Box 73
Lexington, Massachusetts 02173-0073

8. Mr. J. O'Leary 1
Massachusetts Institute of Technology
Lincoln Laboratory
P.O. Box 73
Lexington, Massachusetts 02173-0073

9. Mr. A. Casavant 1
Massachusetts Institute of Technology
Lincoln Laboratory
P.O. Box 73
Lexington, Massachusetts 02173-0073

10. Dr. C. Sequin 1
Associate Chairman
Computer Science Division
University of California
589 Evans Hall
Berkeley, California 94720

11. LTCOL H. W. Carter, USAF 1
Air Force Institute of Technology
Department of Electrical Engineering
AFIT/ENG Building 640 Area B
Wright-Patterson Air Force Base, Ohio 45433

19. Mr. E. Carapezza, Code 52 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943-5000

20. Dr. F.A. Malagon-Díaz 1
2998 Plaza Blanca
Santa Fe, New Mexico 87505-5340

21. CAPT, E. Malagón, USMC 1
SMC #2480
Naval Postgraduate School
Monterey, California 93943

22. LCDR M. A. Malagón-Fajar, USN 1
1220 7th Street, #2
Monterey, California 93940

23. Prof. H. Titus, Code 62Ts 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5000

24. Prof. S. Michaels, Code 62Ms 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5000

25. Prof. L. Abbott, Code 62At 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5000

END

DTIC

9-86